VRIJE UNIVERSITEIT BRUSSEL
FACULTY OF APPLIED SCIENCES
DEPARTMENT OF ELECTRONICS AND INFORMATION PROCESSING

Volumetric Data Compression based on Cube-Splitting and Embedded Block Coding by Optimized Truncation

Xavier Giro

Brussels, 2000

Supervisor : Ir. Peter Schelkens
Promoters: Prof. Jan Cornelis
Prof. Philippe Salembier (UPC)

Thesis submitted to the faculty of Applied Sciences of the Vrije Universiteit Brussel to obtain the degree of Telecommunication Engineer from the UPC

Preface

The work presented in this document was performed in the context of the European Erasmus/Socrates program to which both the Escola Tecnica Superior d'Enginyers de Telecomunicacio de Barcelona (ETSETB) of the Universitat Politecnica de Catalunya(UPC) and the Department of Electronics and Information Processing (ETRO) at the Vrije Universiteit of Brussels (VUB) participate. The thesis was conducted by ir. Peter Schelkens and promoted by Professor Jan Cornelis, both members of the ETRO department, and supervised by Mr. Philippe Salembier from the UPC.

Especially, I would like to thank Mr. Schelkens for his patience, his support and flexibility during these last eight months, as well as the VUB for giving me the opportunity to work at the ETRO department.

But life is not only numbers, computers and work; many people have been of great aid to me. Some of them were very close, like Sylvie, Bea, Marc, Ieia and the whole Erasmus community. Others came to visit me, like Mar, Marta, Ramon, papa, mama, Eva, Albert and Mireia. Others chose to keep electronic contact like Edu, Marti, Neus, Meritxell and half of my family. And some others, like my grandparents and the other half of my family, who have thought a lot about me.

I would like to thank all the people I have mentioned – including those whom I might have forgotten - for giving me the chance to write this document.

0. Introduction

Nowadays, many medical data acquisition devices or multispectral imaging techniques produce three-dimensional image data. These images must be stored in limited space devices or transmitted through limited bandwidth channels. Compression techniques are an extremely valuable tool to reduce the expensive resource requirements.

However, compression techniques have already been developed for the more popular two-dimensional images. Splitting the volumetric image in slices and applying a two-dimensional coding technique to each slice is the philosophy followed by the classical approach for 3D compression. This is clearly inefficient, because 2D techniques only exploit the image correlation in the X and Y axis. In volumetric images a new Z-axis appears, whose correlation must be also exploited to achieve the best results.

The basis for all current image and video compression standards is DCT-based coding. For these techniques the computation is based on splitting of the image into NxN blocks and transforming it from the spatial domain into the DCT domain. Typical examples are first generation coders, like JPEG, which produce a non-structured, unique bit-stream. This technique could easily be adapted to three-dimensional by splitting the volume into NxNxN blocks and applying a 3D DCT. However, one encounters two problems. First, the DCT transform is a lossy, and medical practice cannot tolerate any distortion that could lead to an faulty diagnose. Secondly, contemporary transmission techniques make use of concepts like rate-scalability, quality and resolution scalability, features that are not fully supportable by DCT techniques.

Coders using a wavelet transform as front-end are good candidates to overcome these problems. They scan each bit-planes one by one to generate a structured bit-stream. This bit-stream can be truncated to give more or less quality or resolution, and they are classified second-generation coders. A typical example of 3D wavelet coding is the octave zero-tree based coding [Bil99, Xio99, Kim99, Kim00, Sch00a], which currently tends to deliver the best compression performance. However, it is difficult to control the bit-stream structure since it is dependent on the coder's data flow.

The new image compression standard JPEG2000 uses a third generation technique, called EBCOT ,incorporating an abstract interface to enable reordering of the generated code packages. In this way a fully controllable bit-stream structure is achieved. For example, the bit-stream can be equipped so that resolution or quality scalability are supported. The current verification model (VM7.0) of JPEG2000 however, does not include three-dimensional coding. The only support that is given for multidimensional and/or multi-spectral images is the possibility to execute a wavelet transform along the component axis. Unfortunately, the code supporting this feature was still buggy at the time this document was written

Adapting this third-generation coding technique to a three-dimensional environment was the aim of this thesis. The input volume is transformed into the wavelet transform with the 3D Wavelet front-end described and implemented by Schelkens et al. [Sch00a] and Barbarien [Joeri's thesis]. Later it is coded by an hybrid technique of Cube-Splitting and an JPEG2000's EBCOT module, modified to support the third dimension. The Cube-Splitting module codes big zero-volumes very efficiently, while the EBCOT coder is responsible for the coding of the (sub)volumes containing significant samples. Hence, the implemented coder is called CS-EBCOT.

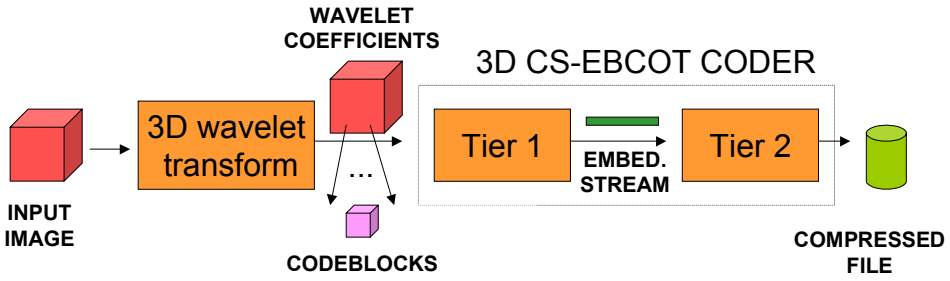


Figure 1: General scheme of the 3D CS-EBCOT coder

The main differences between the 2D JPEG200 and CS-EBCOT are:

- a 3D wavelet transform, which supports more kernels than the current JPEG2000 implementation
- the scanning pattern of the samples into the code-block has been modified to exploit fully 3D relationships
- an extra CS pass has been included in the T1
- 3D contexts have been defined to exploit the correlations between neighbouring voxels along the main spatial axes .
- instead of the MQ-encoder, an alternative adaptive arithmetic coder [10] has been incorparated.
- the tag trees used in the T2 coder exploit the 3D packet structure

1. Technical proposal

1.1 Three dimensional wavelet transform

The first block in any JPEG2000 coder is a wavelet transform. In our case, this transform has had to be adapted to a 3D environment. Some basic principles about wavelets and the filters implemented are given in this section.

1.1.1 Wavelet transforms

The wavelet transform is a popular and powerful mathematical tool, which is commonly being used in contemporary image compression techniques.

Basically, the transforming of an image into the wavelet domain implies the projection of the image onto a set of basis functions. The resulting coefficients, describing the contribution of each of the basis functions, deliver a “decorrelated” image representation. Consequently, most of the signal energy will be contained by the lower subband coefficients, allowing for efficient lossy compression if the corresponding quantization levels are appropriately chosen.

The wavelet transform can be implemented by use of perfect reconstruction finite impulse response filter banks *(Figure 2)*. Basically, the transforming process consists of successively (1) filtering the input image with high-pass (H) and low-pass (L) analysis filters and (2) and a dyadic downsampling of both filter outputs. The inverse transform is performed inverting the scheme and applying the appropriate synthesis filters H' and L'. These filters must satisfy *Equation 1* to enable perfect reconstruction.

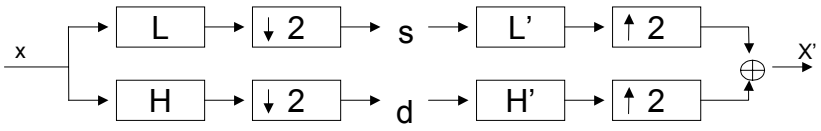


Figure 2: Wavelet transform and Inverse Wavelet transform

(1) CONDITION FOR PERFECT RECONSTRUCTING FILTERS WITH AN I SAMPLE DELAY
$H(z)H'(z) + L(z)L'(z) = 2z^{-l}$ $H(z)H'(-z) + L(z)L'(z) = 0$

Applying the filtering and downsampling steps as many times as desired on the low-pass filter data, realises a multiresolution decomposition as described by Mallat [Mal89]. *Figure 3* shows a two-level Mallat decomposition of a one-dimensional sequence *x*:

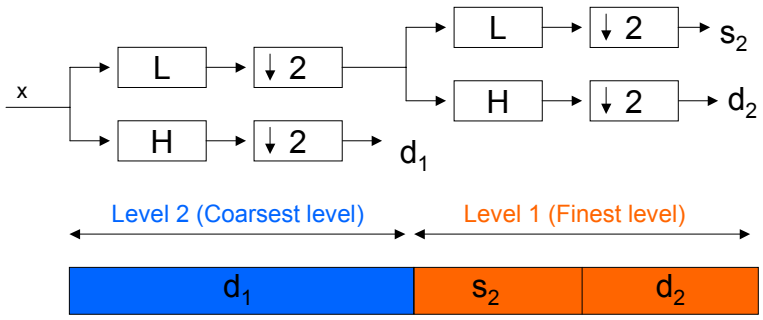


Figure 3: Two-level Mallat decomposition of a one-dimensional sequence

In case the input is a three-dimensional signal, the same filter has to be applied successively in the three spatial directions. However, this is only possible if a separable filter set is used. Otherwise explicit 3D filtering has to be performed. This results in a three-dimensional transformed image that, in the case of a two-level Mallat decomposition, would have the distribution shown in *Figure 4*.

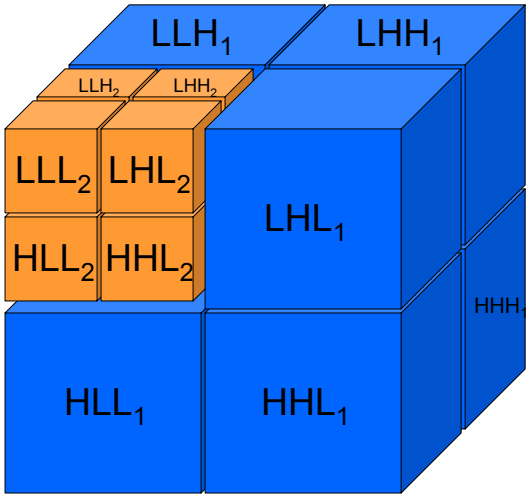


Figure 4: Two-level Mallat decomposition of a three-dimensional signal. H is associated to the high-pass band while L is associated to a low-pass band.

The wavelet transformed volume is thus partitioned into subbands.

Wavelet transforms can be implemented using either floating-point either integer arithmetic. However, floating-point arithmetic does not results in perfect reconstructed image coefficients after the inverse transform in case of a non-quantised wavelet domain. The latter is basically also true for integer arithmetic. However, Wim Sweldens [Swe95] introduced the lifting scheme *(Figure 5)* allowing to compute the discrete wavelet transform with a reduced computational complexity and support for a lossless transform [Cal96]. Lossless techniques are very important for medical images, because a distortion caused by a lossy coding could drive to an erroneous diagnostic. To use this scheme the coefficients $p^{(i)}[k]$ and $u^{(i)}[k]$ must be computed by use of factorisation of a polyphase matrix.

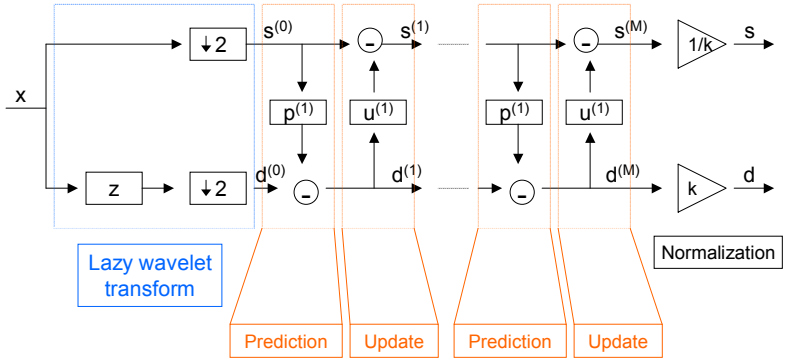


Figure 5: The lifting scheme

This is the scheme is used in the software implementation of the wavelet transform.

Table 1 lists the lossless integer filters being supported by our wavelet transform module. The notation *x/y* indicates that the underlying filter bank has low-pass and high-pass analysis filters of lengths *x* and *y*, respectively.

*Table 1: Lossless integer lifting filters supported by the 3D WT module. The number of vanishing moments is given as (*N,Ñ*), where* N *and* Ñ *specify the number of vanishing moments of the analysis and the synthesis high-pass filters, respectively. The number of filter taps -* l *and* h *- for the low-pass and high-pass analysis filters respectively, is given as* lxh.

Filter Name	Number of Vanishing Moments	Number of Filter Taps	Lifting Steps
5x3 [1]	(2,2)	5x3	$d[n] = x[2n+1] - \lfloor 1/2\,(x[2n] + x[2n+2]) + 1/2 \rfloor$ $s[n] = x[2n] + \lfloor 1/4\,(d[n-1] + d[n]) + 1/2 \rfloor$
S [2]	(1,1)	2x2	$d[n] = x[2n+1] - x[2n]$ $s[n] = x[2n] + \lfloor 1/2\, d[n] \rfloor$
9x7 [1]	(4,2)	9x7	$d[n] = x[2n+1] - \lfloor 9/16\,(x[2n] + x[2n+2]) - 1/16\,(x[2n-2] + x[2n+4]) + 1/2 \rfloor$ $s[n] = x[2n] + \lfloor 1/4\,(d[n-1] + d[n]) + 1/2 \rfloor$
9x3 [1]	(2,4)	9x3	$d[n] = x[2n+1] - \lfloor 1/2\,(x[2n] + x[2n+2]) + 1/2 \rfloor$ $s[n] = x[2n] + \lfloor 19/64\,(d[n-1] + d[n]) - 3/64\,(d[n-2] + d[n+1]) + 1/2 \rfloor$
13x11 [1]	(6,2)	13x11	$d[n] = x[2n+1] - \left\lfloor \begin{array}{l} 75/128\,(x[2n] + x[2n+2]) - 25/256\,(x[2n-2] + x[2n+4]) \\ \quad + 3/256\,(x[2n-4] + x[2n+6]) + 1/2 \end{array} \right\rfloor$ $s[n] = x[2n] + \lfloor 1/4\,(d[n-1] + d[n]) + 1/2 \rfloor$
5x11 [1]	(2+2,2)	5x11	$d^{(1)}[n] = x[2n+1] - \lfloor 1/2\,(x[2n] + x[2n+2]) + 1/2 \rfloor$ $s[n] = x[2n] + \lfloor 1/4\,(d^{(1)}[n-1] + d^{(1)}[n]) + 1/2 \rfloor$ $d[n] = d^{(1)}[n] - \lfloor 1/16\,(-s[n-1] + s[n] + s[n+1] - s[n+2]) + 1/2 \rfloor$
2x6 [3]	(1+1;1)	2x6	$d^{(1)}[n] = x[2n+1] - x[2n]$ $s[n] = x[2n] + \lfloor 1/2\, d^{(1)}[n] \rfloor$ $d[n] = d^{(1)}[n] - \lfloor 1/4\,(s[n-1] - s[n+1]) + 1/2 \rfloor$
S+P [4]	(2,4)	2x6	$d^{(1)}[n] = x[2n+1] - x[2n]$ $s[n] = x[2n] + \lfloor 1/2\, d^{(1)}[n] \rfloor$ $d[n] = d^{(1)}[n] + \lfloor 2/8\,(s[n-1] - s[n]) + 3/8\,(s[n] - s[n+1]) + 2/8\, d^{(1)}[n+1] + 1/2$
13x7 [5]	(4,2)	13x7	$d[n] = x[2n+1] - \lfloor 9/16\,(x[2n] + x[2n+2]) - 1/16\,(x[2n-2] + x[2n+4]) + 1/2 \rfloor$ $s[n] = x[2n] + \lfloor 80/256\,(d[n-1] + d[n]) - 16/256\,(d[n-2] + d[n+1]) + 1/2 \rfloor$

[1] Ali Bilgin, George Zweig, Michael Marcellin "Three-dimensional image compression with integer wavelet transforms", Applied Optics, Vol.39, No.11, pp.1799-1814, 10 April 2000

[2] E. H. Adelson, E. Simoncelli, and R. Hingorani, "Orthogonal pyramid transforms for image coding," in Visual Communications and Image Processing II, T. R. Hsing, ed., Proc. SPIE 845, pp.50-58 (1987).

[3] S. Dewitte and J. Cornelis, "Lossless integer wavelet transform," IEEE Signal Process. Lett. 4, 158-160 (1997).

[4] A. Said and W. Pearlman, "An image multiresolution representation for lossless and lossy compression," IEEE Trans. Image Process. 5, 1303-1310 (1996).

[5] ISO/IEC JTC1/SC29/WG1 WG1N1684.

1.2 CS-EBCOT

The wavelet coeffcient volume is splitted into code-blocks. These code-blocks are processed by an hybrid coder that uses two different coding techniques: a Cube-Splitting(CS) front end and an Embedded Block Coding with Optimised Truncation(EBCOT). The EBCOT is a two-tiered coder (*Figure 6*). The first tier (T1) is the responsible of the block coding. The second tier (T2) organizes the encoded data into a full-featured bit-stream.

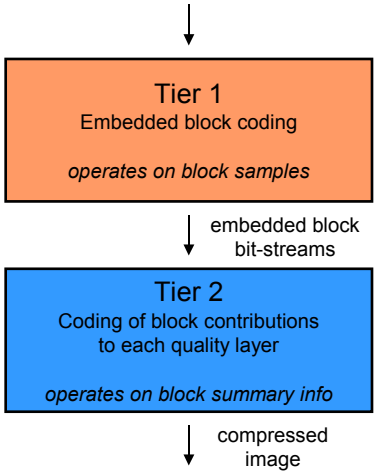


Figure 6: The EBCOT is a two-tiered coder

1.2.1 Code-block partitioning of the wavelet volume

The JPEG2000 standard partitions the wavelet domain into code-blocks, which are going to be coded independently by the EBCOT coder (*Figure 7*).

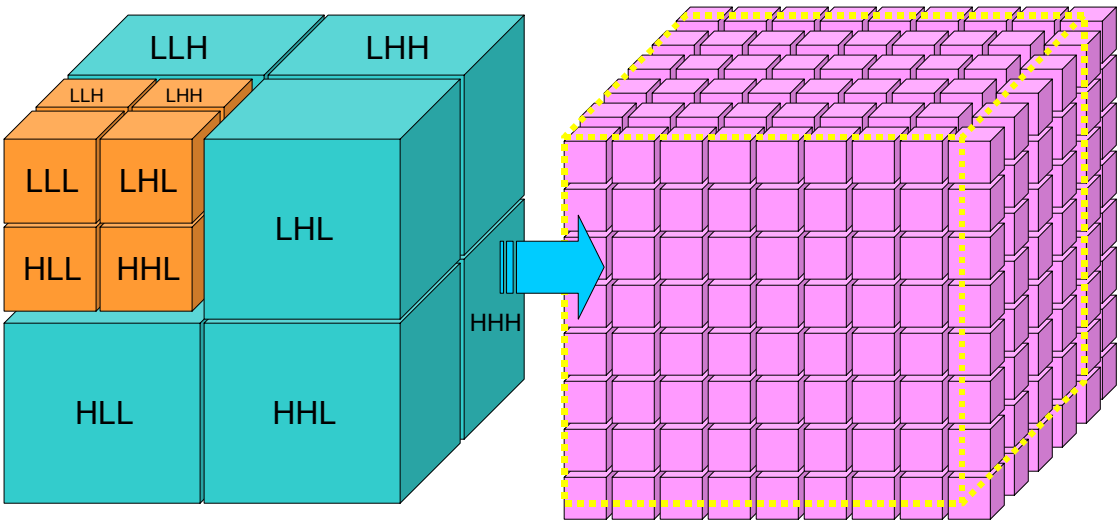


Figure 7: Code-block partitioning of the wavelet volume

The code-block partition can be performed by several ways, so some criterions must be set.

a) THE ORIGIN OF COORDINATES
Despite the JPEG2000 VM 6.0 uses a canvas and partitions the image with tiles, these features have not been included in the described project. In this project, the origin of coordinates is situated on the first sample of the input image, that is, the coordinates (0, 0, 0) of the input image are also the starting point for the code-block partitioning.

b) WHAT TO DO WITH THOSE CODE-BLOCKS WHICH OVERCOME THE IMAGE BOUNDARIES
There is no need why the size of the code-blocks should fit perfectly in the image dimension; in fact, the most probable case is that it doesn't happen. A criterion must be taken to fill in those code-block, which go further than the transformed image.
The criterion taken is to fill those code-blocks with zeros. This is especially interesting because of the octree zero coder implemented in the EBCOT T1, which codes very efficiently a big concentration of zeros.
These added zeros at the code-blocks are removed at the decoder because the decoder knows the original size of the image.

c) HOW TO DEFINE SUBBANDS WHEN AN ODD SIZE MUST BE DIVIDED BY TWO
3D Wavelet transforms splits the transformed image into 8 subbands partitioning every dimension by the half. Besides, this partition can be iterated in the LLL band as many times as depth levels in the wavelet transform.
The criterion followed is that the middle sample of an odd dimension belongs to the low band.

d) WHAT TO DO WITH THOSE CODE-BLOCKS WHICH CONTAIN SAMPLES FROM DIFFERENT SUBBANDS
In some coding primitives used in the EBCOT T1, the subband of the block must be known. As code-block dimension don't have to fit in a unique wavelet subband, it is possible (and probable) that they are going to include samples from different subbands, despite every code-block can be associated to a unique subband.

The criterion taken is that the code-block is going to belong to the same subband to which the first sample of the code-block belongs to.

1.2.2 EBCOT T1

The EBCOT Tier 1 implementation consists of two parts: the Cube-Splitting front-end and the Fractional bit-plane EBCOT coder (*Figure 8*).

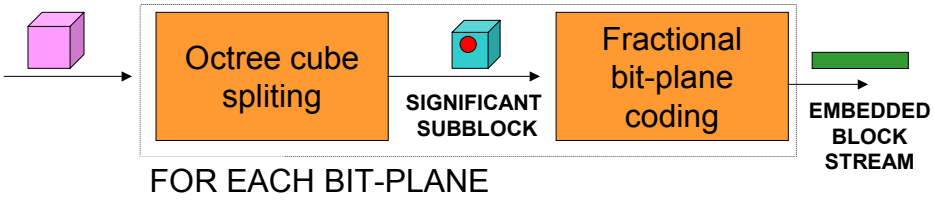


Figure 8: Tier 1

1.2.2.1 Cube-Splitting (CS) front-end

The CS front-end (Qi[p]) is an extra pass not included in the current version of JPEG2000. The 2D version of this coding procedure, called zero coding, was first included in the standard, but was finally rejected because of IP claims by Teralogic Inc. [Chu99] and experiments showed that the coding gain obtained by this coding step was marginal. However, we improved the coding scheme and extended it to 3D [Sch00a], facilitating the exploitation of potential, large area correlations.

The CS front-end is capable of coding large volumes of non-significant samples, which is especially useful at the higher bit-planes where few samples are supposed to be significant. The basic idea is to isolate non-significant bit-volumes with an iterative algorithm.

The CS front-end works on code-blocks. The algorithm is the following:

1) Obtain the code-block to be coded.
2) Check the significance of the current (sub-)block : this is evaluated by checking all the samples in the (sub-)block against the current threshold. If (a) sample(s) is (are) significant, encode the SGN symbol and go to step 3. If there is no significant sample for the current threshold, a NSG symbol is encoded and step 2 is repeated for the next sub-block at the same level until all sub-blocks are encoded. Then, the sub-blocks at the level above are considered (if needed), otherwise coding is ended. Remark that this a depth-first scanning approach.

3) If the (sub-)block size is larger than the user-defined, minimal allowed sub-block size, repartition the significant (sub-)blocks in 8 equally-sized sub-blocks and go back to step 2.

However, unlike the proposal in [Sch00a] no (context-based) arithmetic coding is issued. Additionally, during encoding one is only refining the nodes that have been identified as non-significant for previous thresholds.

A simple example, where the only significant sample is situated in the origin, is given in *Figure 9*.

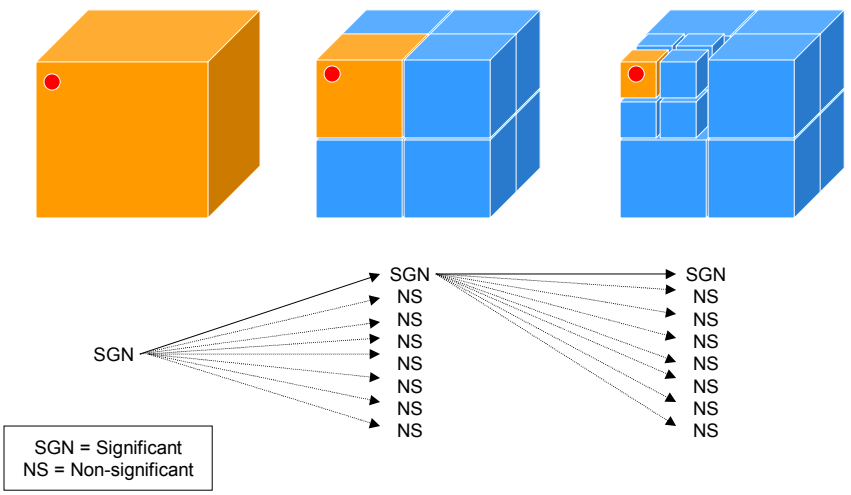


Figure 9: Example of Cube-Splitting coding technique

Doing that and setting the minimum size to which the cube could be spliced to 1x1x1 would create a complete coding algorithm. Some techniques related to these principles are discussed in [Sch99a-c].

As mentioned, the current implementation reuses the CS information coded in the higher bit-planes for the encoding of the lower bit-planes since significant sub-blocks will be encoded with the fractional bit-plane coder for all lower thresholds.

1.2.2.2 Fractional bit-plane coding

1.2.2.2.1 Outline

The fractional bit-plane coder encodes only the sub-blocks that have been identified as significant by the Cube-Splitting coder. Three passes are defined per bit-plane: the forward significance propagation pass, the magnitude refinement pass and the normalization pass. They three coding passes are defined in ordered in such a way that most relevant data is encoded first, consequently generating potential truncation points in the bit-stream.

Additionally, these coding pass use several coding operations (primitives), i.e. the zero coding (ZC), sign coding (SC), magnitude refinement (MR) and run-length coding (RLC) primitives. These primitives enable the selection of suitable context models for the subsequent arithmetic coding or run-length coding stages.

The state of each code-block sample is stored in four state variable arrays and consulted and updated during the different coding passes en operations:

- The significance state variable array σ_i [m,n,o] identifying significant voxels (initialised to 0)
- The visited state variable array η_i [m,n,o] identifying already encoded voxels for the current bit-plane (initialised to 0 when the encoding of a new bit-plane starts)
- The refined state variables array δ_i[m,n,o] identifying voxels already refined with the MR primitive in a previous bit-plane(initialised to 0)
- The sign state variables array χ_i[m,n,o] recording the sign of each significant voxel (0 for positive, 1 for negative)

1.2.2.2.2 Scanning pattern

The scanning order of the coefficients in the code-block is not unimportant for the coding performance. The arithmetic coder performs better when there are no big changes in the statistical distribution of the data being coded. To achieve this, big jumps in the scanning order of the coefficients must be avoided because data tends to be the more homogeneous the closer they are.

The CS-EBCOT software supports two scanning patterns.

1.2.2.2.2.1 Expansion of 2D JPEG2000 scanning pattern to a volumetric slice by slice scanning pattern

The first scanning pattern is identical to its 2D version for each slice, and extended in a slice-by–slice manner. The samples are read in groups of four vertically aligned samples. When a complete slice is stripe-wise processed, the subsequent slice is processed (in the z-direction), as shown in *Figure 10*.

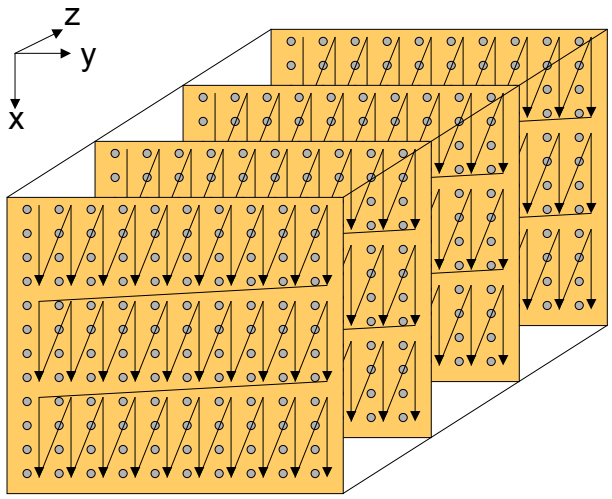


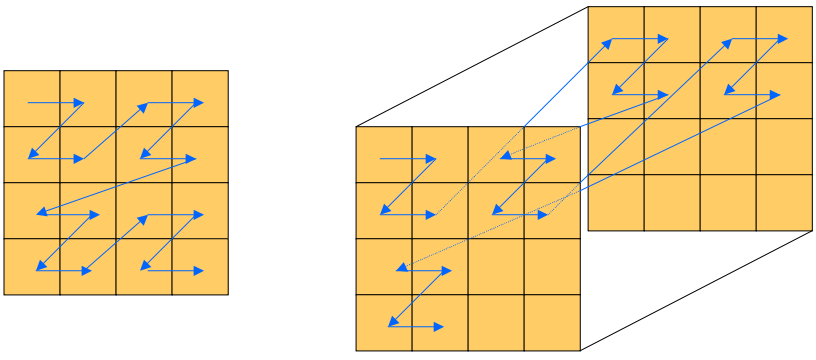
Figure 10: Expansion of 2D JPEG2000 scanning pattern

1.2.2.2.2.2 Morton scanning pattern

A new scanning pattern has been included to exploit the third dimensional correlation. It is a 3D extension of the well-known Morton space-filling curve [Mor66] and can be easily generated with a binary counter by associating its bits to one of the three spatial coordinates (see *Table 2* and *Figure 11*).

Table 2: three-dimensional Morton scanning pattern

Binary counter						Morton coordinates		
z_2	y_2	x_2	z_1	y_1	x_1	x	y	z
0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	0	0
0	0	0	0	1	0	0	1	0
0	0	0	0	1	1	1	1	0
0	0	0	1	0	0	0	0	1
0	0	0	1	0	1	1	0	1
0	0	0	1	1	0	0	1	1
0	0	0	1	1	1	1	1	1
0	0	1	0	0	0	2	0	0
0	0	1	0	0	1	3	0	0
0	0	1	0	1	0	2	1	0
0	0	1	0	1	1	3	1	0
0	0	1	1	0	0	2	0	1
0	0	1	1	0	1	3	0	1
0	0	1	1	1	0	2	1	1
0	0	1	1	1	1	3	1	1
0	1	0	0	0	0	0	2	0



2D Morton

3D Morton

Figure 11: 2D and 3D Morton Scanning pattern

1.2.2.2.3 Coding passes

1.2.2.2.3.1 Significance propagation pass

As mentioned earlier three coding passes are executed for each bit-plane (except the first one). To code a pixel with the significance propagation pass ($P_i[p,1]$), it must have been classified as non-significant (σ_i [m,n,o] =0) and have at least one significant pixel in its preferred neighbourhood (σ_i [m,n,o]=1). The preferred neighbourhood refers to the twenty-six voxels around the voxel being coded. Next, the Zero Coding (ZC) primitive is activated and, if a new significant pixel is identified, ZC primitive calls the Sign Coding (SC) primitive. It also sets η_i [m,n,o] = 1 to mark that the sample has been coded for the current bit-plane to avoid redundant information embedding (*Figure 12*).

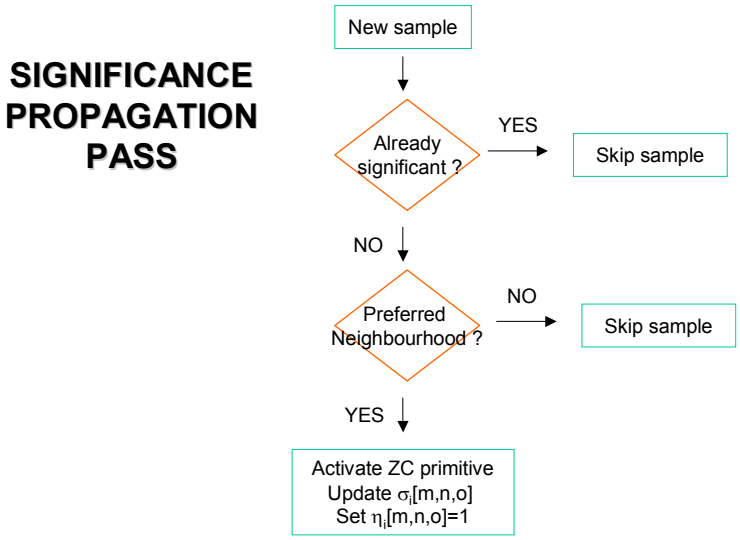


Figure 12: Significance propagation pass algorithm

1.2.2.2.3.2 Magnitude refinement pass

The magnitude refinement pass ($P_i[p,2]$) encodes only those samples that have been marked significant in previous bit-planes (η_i [m,n,o] = 0 and σ_i [m,n,o] =1). It uses the MR primitive to encode the refinement bits (*Figure 13*).

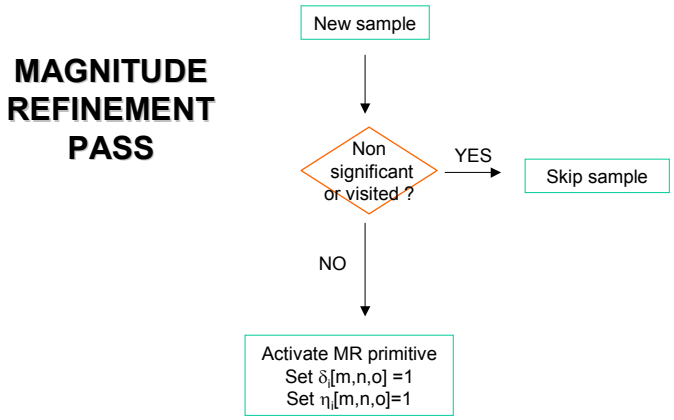


Figure 13: Magnitude refinement pass algorithm

1.2.2.2.3.3 Normalization pass

This normalization pass (Pi[p,3]) looks for the new significant pixels with considering a preferred neighbourhood (i.e. significance as in the significance propagation pass is not required from the surrounding samples). It visits only those samples with η_i [m,n,o] = 0 and σ_i [m,n,o] =0 and uses the ZC, RLC and SC primitives. This pass can be understood as a garbage collector, because it processes all these samples that had not been visited in any of the previous passes (*Figure 14*).

At the end of this pass all η_i [m,n,o] are set to 0 as a preparation for the next step.

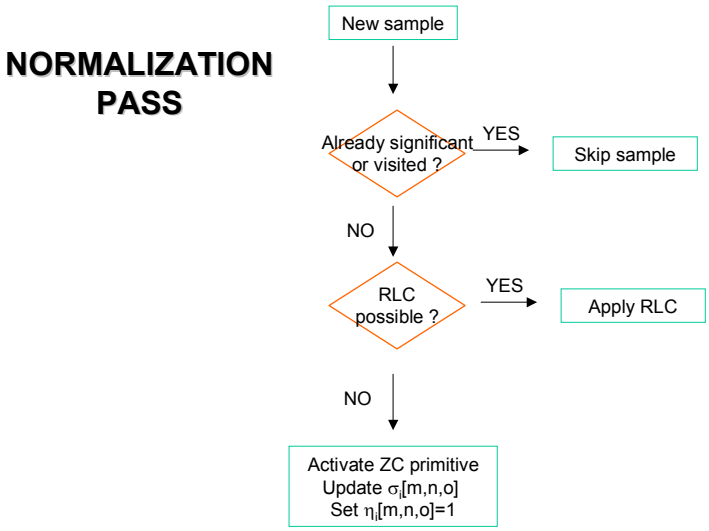


Figure 14: Normalization pass algorithm

1.2.2.2.3.4 Order of the coding passes

A defined order of the three coding passes is followed in each bit-plane. Firstly the Significance Pass, then the Magnitude Refinement Pass and finally the Normalization Pass. The main philosophy of this order is refine first the already identified spatial structures in the image by adding extra points to it, before introducing new isolated structures (edges). Basically, we follow a morphological approach. An example for a two-dimensional image is shown in the next *Figure 15*.

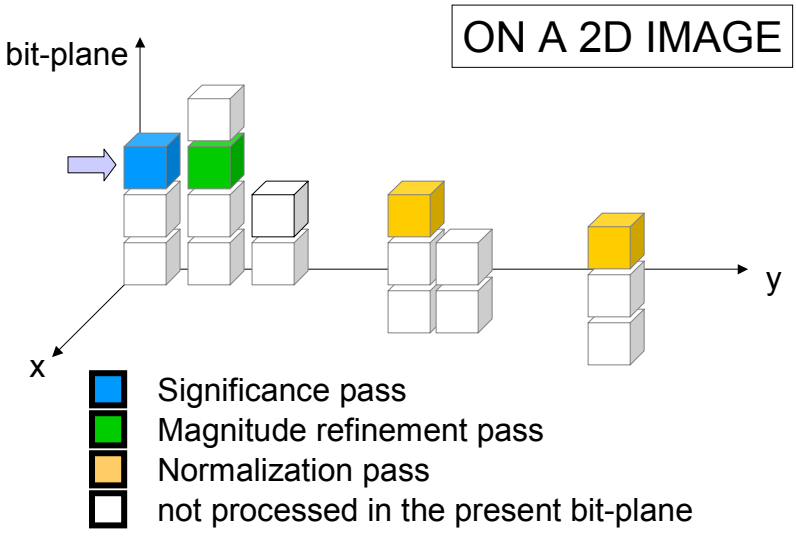


Figure 15: Coding passes on a two-dimensional image

On the first bit-plane the Significance and Magnitude Refinement passes are skipped, as there are no significant samples marked from previous planes. In that case, the first pass performed is the Normalization pass.

1.2.2.2.4 Coding operations

Four primitives are defined to support the encoding process in the different coding passes. Each coding primitive has got its own look-up table to identify the probability model that has to be issued by the arithmetic coder for a given context situation.

The context situation is identified based on the condition of the neighbouring samples. The neighbouring samples can be classified depending on the distance to the sample being processed. While nine samples were taken into account in the 2D implementation, the 3D version considers twenty-six neighbours (*Figure 16*). This is the biggest difference between the 2D JPEG2000 and CS-EBCOT. This explains why more contexts have been defined

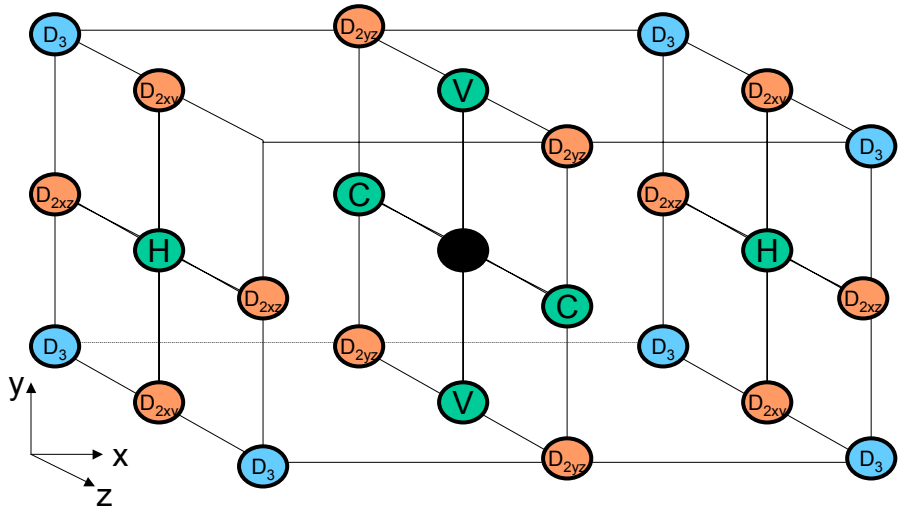


Figure 16: Indexing of neighbouring samples

The context is also influenced by the subband, the considered wavelet coefficient is belonging to. Knowing to what subband belongs the sample being coded is important for the coder, because it can give some important information for the arithmetic coder. For example, in the LLL subband, a sample which is between two significant samples, is more probable to be significant than if we have got the same situation in the HHH subband. The 3D JPEG2000 coder makes use of this information to achieve a better behaviour of the arithmetic coder.

The four coding operations are:

- ZERO CODING (ZC)

This primitive codes, dependent on the significance states of the surrounding pixels, the significance state for the specified voxel bit in the examined bit-plane.

If A = {all immediate neighbours nodes}, X = {(x,y,z) $\in$ A | y=z=0}, Y= {(x,y,z) $\in$ A | x=z=0} and Z = {(x,y,z) $\in$ A | x=y=0} s (all three passing through the origin node = voxel to be encoded) then the following sets can be identified (*Figure 16)*.

- H = A $\cap$ X
- V = A $\cap$ Y
- C = A $\cap$ Z
- D_{2xy} = {(x,y,z) $\in$ A | abs(x)=abs(y) and z=0}
- D_{2xz} = {(x,y,z) $\in$ A | abs(x)=abs(z) and y=0}
- D_{2yz} = {(x,y,z) $\in$ A | abs(y)=abs(z) and x=0}
- D_3 = {(x,y,z) $\in$ A | abs(x)=abs(y)=abs(z)}

To build the context look-up tables (*Tables 3, 4, 5 and 6)* the three spatial direction x, y and z were understood as high or low pass band directions. The context assignment depends on the significance of the neighbouring samples, not for the relative XYZ position to the sample being coded but for the low or high band filtering performed on each direction. For example, it is more probable to have two consecutive samples on the X direction than in Y or Z if the sample is located in the LHH subband. This was already the criterion followed in the 2D implementation of JPEG2000.

Table 3: Look-up table for LLL subband in ZC

LLL subband			
H+V+C	$D_{2xy}+D_{2xz}+D_{2yx}$	D_3	context
≥4	x	x	15
3	x	x	14
2	≥1	x	13
2	0	≥1	12
2	0	0	11
1	≥2	x	10
1	1	≥1	9
1	1	0	8
1	0	≥1	7
1	0	0	6
0	≥2	≥1	5
0	≥2	0	4
0	1	≥1	3
0	1	0	2
0	0	≥1	1
0	0	0	0

Table 4: : Look-up table for LHL subband in ZC

LHL subband					
H + C	V	D_{2xy}	$D_{2xy} + D_{2yz}$	D_3	context
≥2	x	X	x	x	15
1	≥1	X	x	x	14
1	0	≥1	x	x	13
1	0	0	≥1	x	12
1	0	0	0	≥1	11
1	0	0	0	0	10
0	2	X	x	x	9
0	1	X	x	x	8
0	0	≥2	x	x	7
0	0	1	≥1	x	6
0	0	1	0	≥1	5
0	0	1	0	0	4
0	0	0	≥1	≥1	3
0	0	0	≥1	0	2
0	0	0	0	1	1
0	0	0	0	0	0

(also valid for HLL and LLH subbands)

Table 5: : Look-up table for HLH subband in ZC

HLH subband					
H + C	V	D_{2xz}	$D_{2xy} + D_{2xz}$	xyz	context
x	2	x	x	x	15
x	x	≥3	x	x	14
≥1	1	2	x	x	13
0	1	2	x	x	12
0	1	1	x	x	11
≥1	1	1	x	x	10
≥1	1	0	x	x	9
0	1	0	x	x	8
≥1	0	≥2	x	x	7
≥1	0	1	x	x	6
0	0	≥1	x	x	5
≥2	0	0	x	x	4
1	0	0	≥1	x	3
1	0	0	0	x	2
0	0	0	0	≥1	1
0	0	0	0	0	0

(also valid for HHL and LHH subbands)

Table 6: : Look-up table for HHH subband in ZC

HHH subband			
H + V + C	$D_{2xy} + D_{2xz} + D_{2yz}$	D_3	context
≥4	X	x	15
3	X	x	14
2	≥1	x	13
2	0	≥1	12
2	0	0	11
1	≥2	x	10
1	1	≥1	9
1	1	0	8
1	0	≥1	7
1	0	0	6
0	≥2	≥1	5
0	≥2	0	4
0	1	≥1	3
0	1	0	2
0	0	≥1	1
0	0	0	0

(is the same that for the LLL)

- SIGN CODING (SC)

When a new sample becomes significant the sign is coded using the context that is identified by the average signs of the H, V and C sets. Additionally, a predictor is defined based on the sign information of those sets. If the predicted sign is the same as the real one, a 0 symbol is coded, a 1 otherwise.

The prediction of signs is done following the next table, which has been modified from the 2D implementation. To do that, an array $\chi_i[m,n,o]$ saves the sign information of all coded samples, having a 0 for positive samples and 1 for negative samples.

The meaning of the figures of *Table* 8 can be found in *Table 7*:

Table 7: Meaning of figures in SC context look-up table

1	- both neighbours are significant and positive
0	- both neighbours are insignificant - both neighbours are significant but have opposite sign
-1	- one or both neighbours are significant and negative

$\hat{\chi}$ is the sign predictor

The context table was generated starting from the following principles:

- 0 values don't give any information about the sign
- 1 values are 'stronger' than –1 values
- 1 values are associated to positive samples while –1 values are associated to negative samples

Table 8: : Look-up table for ZC

H	V	C	$\hat{\chi}$	context
1	1	1	0	5
1	1	0	0	4
1	1	-1	0	3
1	0	1	0	4
1	0	0	0	1
1	0	-1	0	2
1	-1	1	0	3
1	-1	0	0	2
1	-1	-1	1	3
0	1	1	0	4
0	1	0	0	1
0	1	-1	0	2
0	0	1	0	1
0	0	0	0	0
0	0	-1	1	1
0	-1	1	0	2
0	-1	0	1	1
0	-1	-1	1	4
-1	1	1	0	3
-1	1	0	0	2
-1	1	-1	1	3
-1	0	1	0	2
-1	0	0	1	1
-1	0	-1	1	4
-1	-1	1	1	3
-1	-1	0	1	4
-1	-1	-1	1	5

- MAGNITUDE REFINEMENT (MR)

This primitive codes the bit value in the present bit plane of an already significant sample. It consults the $\delta_i[m,n,o]$ state array. It identifies whether the magnitude refinement primitive has already been applied to the sample in a previous bit-plane. No new contexts have been defined compared to the original EBCOT implementation (*Table 9*).

Table 9: : Look-up table for MR

$\delta_i[m,n]$	H + V + C	context
1	x	2
0	≥ 1	1
0	0	0

- RUN-LENGTH CODING (RLC)

The RLC primitive (*Figure* 17) is applied in conjunction with the ZC primitive to reduce the average number of binary symbols, which must be encoded during the normalization pass. The primitive is invoked in place of the ZC primitive if and only if the following three conditions are fulfilled:

a) 4 consecutive samples must have zero state variable $\sigma_i[m,n,o] = 0$.
b) all four samples must have identically non-significant neighbourhoods.
c) the group of samples must be aligned on a four-sample boundary within the scan, because there is a fixed scanning pattern at each pass of four samples. That means that the group of four samples must be exactly a column of the column-based stripe-scanning pattern.

When a group of four symbols satisfying the above conditions is encountered, a single symbol is encoded to identify whether any sample in the group is becoming significant in the current bit plane (1 if so, 0 otherwise). This symbol is coded using a single arithmetic coding context state.

If any of the four symbols becomes significant, the zero-based index of the first significant sample in the group is sent as a two-bit quantity. The most significant bit is sent first followed by the least significant bit. Both are sent using MQ coder's UNIFORM context, whose associated probability model is intended to reflect a uniform distribution.

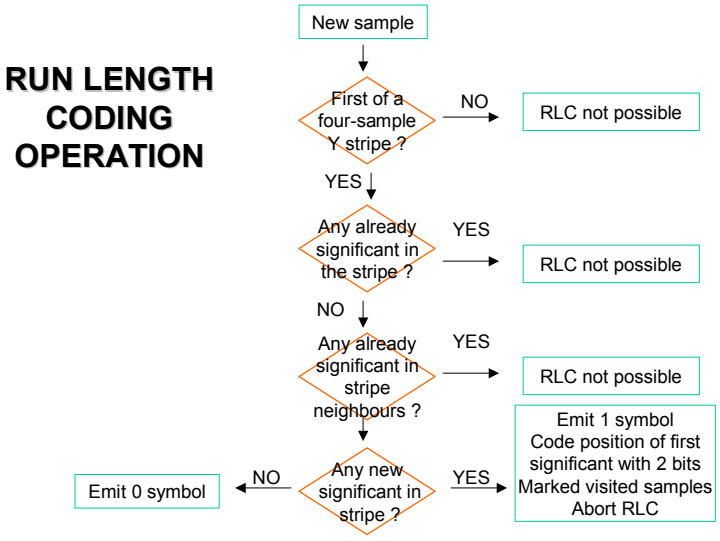


Figure 17: Run Length Coding Operation

1.2.2.2.5 Efficient distortion estimation for R-D optimal truncation

In EBCOT's second tier (T2), the information is coded in quality layers. The process that identifies the different quality levels for each code-block is based on a rate-distortion estimation, which is computed for and associated to every pass during the T1 coding.

The candidate truncation points for the embedded bit-stream of each code-block are recorded at the conclusion of each coding pass. During compression, the number of bytes R_i^n, required to represent all coded symbols up to each truncation point n, as well as the distortion D_i^n, incurred by truncating the bit-stream at each point, n, is assessed.

Truncating the bit-stream by only coding the most significant bits (higher bit-planes) is often referred to as implicit quantization. The quantization step-size Δ_i is than attributed a value equal to 2^p, where p is the index of the last available bit-plane.

1.2.2.2.5.1 Non-Reversible Transforms

The rate-distortion optimisation algorithm, based on Lagrangian rate-allocation (LRA) [Vm_60], depends only on the amount by which each coding pass reduces distortion. Specifically, if D_i^0 denotes the distortion incurred by skipping the complete code-block, then we need only to compute the differences D_i^q - D_i^{q-1}, for q=1, 2, 3…(q indicating the different truncation points). This computation can be performed with the aid of two small lookup tables, which do not depend upon the coding pass, bit-plane or subband involved. This can be demonstrated as follows:

Δ_i : quantization step size in block B_i

$v_i[m,n,o]$: magnitude representation of the sample.

$\omega_i \Delta_i^{\,2}$: contribution to the distortion in the reconstructed image which would result from an error of exactly one step size in a single sample from code-block B_i

ω_i : positive weight which is computed from the L2 norm of the relevant subband's wavelet synthesis and may, additionally, be modified to reflect visual weighting or other criteria.

$\bar{v}_i^{\,p}[m,n,o]$: normalized difference between the magnitude of sample $s_i[m,n,o]$ and the largest quantization threshold in the previous bit-plane which was not larger than the magnitude.

$$\bar{v}_i^{\,p}[m,n,o] = 2^{-p} v_i[m,n,o] - 2\left\lfloor \frac{2^{-p} v_i[m,n,o]}{2} \right\rfloor$$

When a single sample first becomes significant in a given bit-plane p, we must have $v_i[m,n,o] \geq 2^p$ and hence $\bar{v}_i^{\,p}[m,n,o] \geq 1$ and the reduction in distortion is expressed in *Equation 2*.

(2) REDUCTION IN DISTORTION WHEN A SINGLE SAMPLE FIRST BECOMES SIGNIFICANT

$$2^{2p}\omega_i{\Delta_i}^2\left(\left(\bar{v}_i^{\,p}[m,n,o]\right)^2 - \left(\overline{v_i^p[m,n,o]} - 1.5\right)^2\right) = 2^{2p}\omega_i\Delta_i^2 \cdot f_s\left(\bar{v}_i^{\,p}[m,n,o]\right)$$

Provided that of course the representation levels used during inverse quantization are midway between the quantization thresholds, which is the case in our implementation. Also, the reduction in distortion, which may be attributed to magnitude refinement of a sample in the bit-plane p, it is expressed in *Equation 3*.

(3) REDUCTION IN DISTORTION WHEN MAGNITUDE REFINEMENT

$$2^{2p}\omega_i{\Delta_i}^2\left(\left(\bar{v}_i^{\,p}[m,n,o]-1\right)^2 - \left(\overline{v_i^p[m,n,o]} - 0.5\right)^2\right) = 2^{2p}\omega_i\Delta_i^2 \cdot fm\left(\bar{v}_i^{\,p}[m,n,o]\right)$$

Thus, the reduction of distortion incurred during a single coding pass may be computed by summing the outputs of one of the two different function $f_s(\cdot)$ or $f_m(\cdot)$, depending on the coding pass, and then scaling the result at the end of the coding pass by a constant value which is easily computed from the bit-plane index and the value of ${\omega_i\Delta_i}^2$.

1.2.2.2.5.2 Reversible Transforms

In general, the process for distortion estimation is the same for non-reversible and reversible transforms, but there are two subtle differences that must be pointed out here.

The equations showed in the previous section are based upon the assumption that the inverse quantisation will represent each coefficient with the mid-point of the relevant quantization interval. This is the most likely behaviour for the quantisation most of the time, except for the least significant bit-plane in the reversible mode. In this case there is no quantization error and midpoint reconstruction has no sense.

In this case, the previous *Equations (2)* and *(3)* should be by *Equations (4)* and *(5)*.

(4) REDUCTION IN DISTORTION WHEN A SINGLE SAMPLE FIRST BECOMES SIGNIFICANT WHEN CODING THE LEAST SIGNIFICANT BIT

$$2^{2p}\omega_i{\Delta_i}^2\left(\bar{v}_i^{\,p}[m,n,o]\right)^2 = 2^{2p}\omega_i\Delta_i^2 \cdot f'_s\left(\bar{v}_i^{\,p}[m,n,o]\right)$$

(5) REDUCTION IN DISTORTION WHEN MAGNITUDE REFINEMENT WHEN CODING THE LEAST SIGNIFICANT BIT

$$2^{2p}\omega_i{\Delta_i}^2\left(\left(\bar{v}_i^{\,p}[m,n,o]-1\right)^2\right) = 2^{2p}\omega_i\Delta_i^2 \cdot f'_m\left(\bar{v}_i^{\,p}[m,n,o]\right)$$

1.2.2.2.5.3 Software implementation

The software implementation to compute the reduction in distortion uses two small lookup tables that are created at the beginning of the T1 Coding.

The argument to theses functions, $\bar{v}_i^p[m,n,o]$, has a binary representation of the form v.xxxxx, where v, the only bit before the binary point, is simply the value of the magnitude bit p. In the present implementation, exactly 6 extra bits beyond the binary point are used to index a 7-bit lookup table for $f_m(\cdot)$ and a 6-bit lookup table for $f_s(\cdot)$. Each entry of these lookup tables holds a 16-bit fixed point representation of $2^{13} f_s(\cdot)$ or $2^{13} f_m(\cdot)$

1.2.3 EBCOT T2

For each subblock B_i, a separate bit-stream has been generated without using any information from the other blocks. Moreover, the bit-stream has the property that it can be truncated in several potential truncation points. The Tier 2 part of EBCOT is taking care of the truncation of the bit-streams to reach the desired bit-rate while minimizing the introduced distortion.

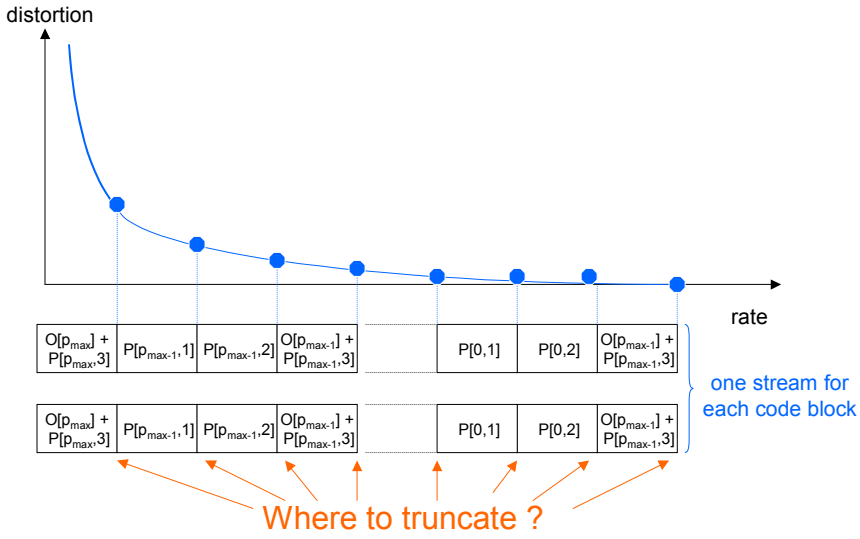


Figure 18: Multiple truncation points in each bit-stream with an associated distortion

Figure 18 shows that the end of each coding pass is a potential truncation point for each bit-stream generated from a code-block. The graph states that truncating the bit-stream at a high bit-plane introduces more distortion than doing it at a lower bit-plane. Besides, the distortion associated to each coding pass is different from code-block to code-block. Hence, choosing the appropriate truncation point for each code-block bit-stream is not an easy decision and requires some simulation or computation.

1.2.3.1 Tag trees

Before discussing what information needs to be added in the T2, a simple tree structure is going to be introduced. The 'tag tree' is a particular type of tree structure, which provides the framework for efficiently exploit the redundancy between different code-blocks from the same subband and between different bit-stream layers.

The basic idea of the tag tree concept is to build a tree whose leaves correspond to code-blocks. The quantities to be encoded are associated with every leaf. The coding process is driven from the leaves to the root by grouping leaves in blocks of 2x2x2. The information

associated with every non-leaf node is a minimal set mutual to all descendent nodes. The process is repeated until the root node is reached.

1.2.3.1.1 Tag tree algorithm T(m,n,o,t)

Let $q_1[m,n,o]$ denote a three-dimensional array of quantities which we would like to represent via the tag tree. Let $q_2[m,n,o]$ denote the nodes at the next level of the oct-tree structure. Each of these nodes is associated to a 2x2x2 code-block structure, expect those which are at the boundaries. Let denote the root node as $q_k[0,0,0]$

The purpose of the tag tree algorithm s to encode the minimum amount of information necessary to identify whether or not $q_1[m,n,o] \geq t$. More precisely, we express the tag tree algorithm as a procedure T(m,n,o,t) which encodes the minimal amount of information required to indicate whether or not $q_1[m,n,o] \geq t'$ for each t' in the range $1 \leq t' \leq t$. This is exactly what we need to code and decode the packets in the T2.

To assist in the encoding process two extra variables are going to be used:

- $t_k[m,n,o] \Rightarrow$ represents the information which has currently been encoded concerning the value of $q_k[m,n,o]$. During the coding process $t_k[m,n,o]$ increases monotonically. It is initialised to zero. $q_k[m,n,o]$ is completely identified once $t_k[m,n,o] > q_k[m,n,o]$.
- $t_{min} \Rightarrow$ it is used to propagate knowledge from ancestor nodes to their descendants. It is the greatest lower bound on the most recent ancestor's q value.

The algorithm invoked is as follows (*Figure 19*):

1. set k = K, i.e. start at the root node; we will take a linear path from the root toward the leaf node identified by the indices m, n and o.
2. set $t_{min} = 0$.
3. set $m_k = \lfloor m / 2^{k-1} \rfloor$, $\lfloor n / 2^{k-1} \rfloor$ and $\lfloor o / 2^{k-1} \rfloor$, so that $[m_k, n_k, o_k]$ is the location of the leaf node's ancestor in level k.
4. if $t_k[m_k,n_k,o_k] < t_{min}$ then the state variable $t_k[m,n,o]$ is out of date; set $t_k[m_k,n_k,o_k] = t_{min}$.
5. if $t \leq t_k[m_k,n_k,o_k]$
    - if k = 1, we have reached the leaf and $t_k[m_k,n_k,o_k] = t_1[m,n,o]$. By definition of our state variable, sufficient information must have been encoded to identify whether or not $q_1[m,n,o] \geq t$ for each $t' \leq t_1[m,n,o]$ and $t \leq t_1[m,n,o]$, so the algorithm is finished.
    - Otherwise
        - Update the value of t_{min} in preparation of moving to the next lower level in the tree; recall that t_{min} is interpreted as the greatest lower bound on the most recent ancestor's q value which can be deduced from what has been encoded. The current node is the most recent ancestor for the node we will visit in the next level, so set $t_{min} = \min \{ t_k[m_k,n_k,o_k], q_k[m_k,n_k,o_k] \}$.
        - Set k = k – 1 and go to step 3.

6. Otherwise, i.e. if $t > t_k[m_k,n_k,o_k]$, then send enough information to increment $t_k[m_k,n_k,o_k]$, Specifically,

- if $q_k[m_k,n_k,o_k] > t_k[m_k,n_k,o_k]$, emit a '0' bit.
- Else, if $q_k[m_k,n_k,o_k] = t_k[m_k,n_k,o_k]$, emit a '1' bit.
- Set $t_k[m_k,n_k,o_k] = t_k[m_k,n_k,o_k] + 1$
- Go to step 5.

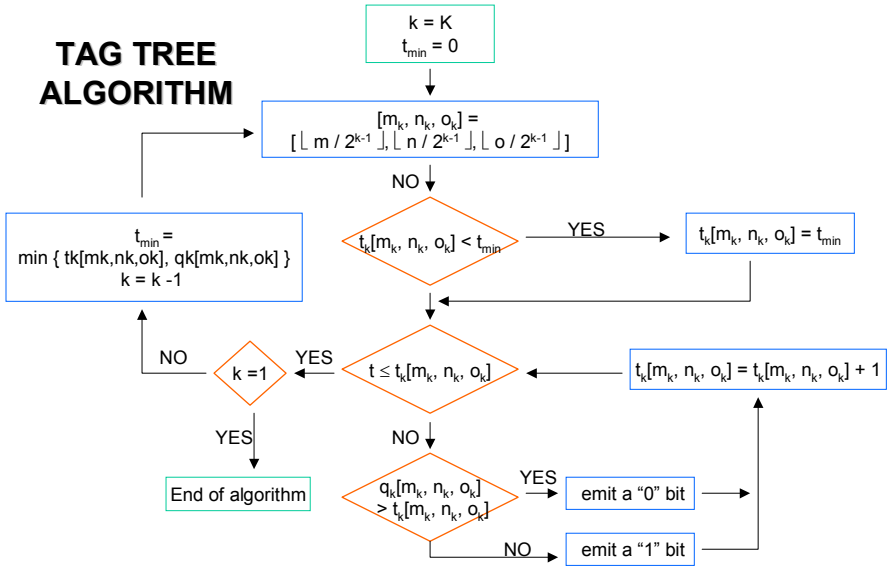


Figure 19: Tag tree algorithm

1.2.3.2 General characteristics of EBCOT T2

The bit stream is organised in quality layers, labelled λ = 1, 2, … Λ, which can be scaled with respect to resolution and number of image components.

Each layer contains a separate packet for each resolution level (l = 0, 1, .. L) and each image component. $K_\lambda^{l,c}$ contains the new information from subbands in resolution level l, for image component c, which is being introduced in the bit stream layer λ. In our implementation, only one component is supported.

Ae packet is associated to each resolution level at each quality layer. Each packet contains a header and a body. The header contains information related to the code-blocks whose compressed stream is included in the body of the packet.

HEADER
• code-block inclusion information for bit-stream layer λ. • the maximum bit-depth for all code-blocks being included in the bit-stream for the first time. • the number of new coding passes which are being included for each included code-block • the number of new code bytes which are being included from the embedded code-block streams.
BODY
• code-block bytes by themselves

The packets are always byte aligned, but explicit markers do not align a packet's head and body. No arithmetic coder is used when coding the headers of the packets.

The packets are written following a progressive by SNR criteria. This means that all the packets of the lowest bit-stream layer are sent first, later the ones from a higher layer. In every bit-stream layer level we can find different packets for each resolution, ordered from lower to higher resolutions. For one given resolution, the order of appearance of the subbands must be established. We start with the lowest resolution bands. For a given resolution, first the LLL band is transmitted, followed by the HLL, LHL, HHL, LLH, HLH, LHH and HHH subbands (*Figure 20*).

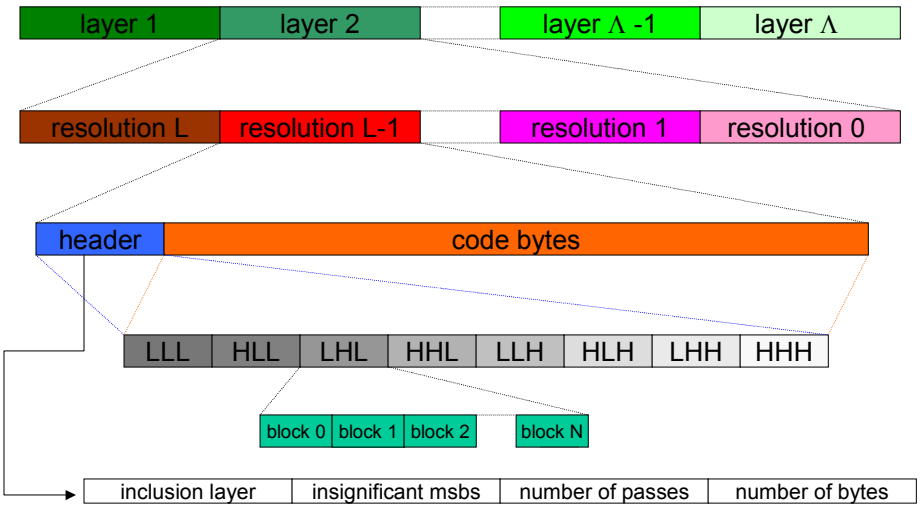


Figure 20: Data distribution in compressed file

Another option would have been progressive by resolution. All the packets of the lowest resolution are then sent first, and later the ones from a higher resolution. Hence, in every resolution level we can find the different packets ordered by quality. This capability is not implemented in the current version of the codec.

1.2.3.3 Anatomy of the packet header

1.2.3.3.1 Inclusion information

A tag tree is used to efficiently represent the bit-stream layer in which a code-block is included for the first time.

The quantities at the leaves of the tag tree are set to the index of the bit-stream layer in which the code-block is first included minus 1.

For any given code-block, the inclusion information is represented in one of two different ways, depending whether or not the block has been included in a previous bit-stream layer.

a) IF IT HAS BEEN INCLUDED IN PREVIOUS BIT-STREAM LAYER ($\lambda[m,n,o] < \lambda$)
   We simply send a single bit to identify whether or not any new information for the code-block is included at the current bit layer (1 if new info is included, 0 otherwise)
b) IF IT HAS NOT YET BEEN INCLUDED IN ANY PREVIOUS BIT-STREAM LAYER
   We invoke tag tree coding procedure T(m,n,o, λ).

In practice, we can and do exploit the fact that the quantities associated with nodes in the tag tree can be updated after the coding has started, provided the update satisfies the conditions necessary for a tag tree encoding. This means that we do not need to determine the value of $\lambda[m,n,o]$ for all code-blocks in the subband ahead of time. Instead we initialise the values stored at each node to infinity (or a suitable large integer).

When we determine that a code-block should be first included in the layer based upon the available rate-distortion information, then we just set the relevant quantity $q_1[m,n,o] = \lambda[m,n,o] - 1$ to the tag tree lave.

1.2.3.3.2 Maximum bit-depth information

Let $p^{max}[m,n,o]$ denote the index of the most significant bit-plane with respect to which any sample in the code-block B[m,n,o] is significant. The number of magnitude bit-planes available for the relevant subband is given by M_b so that $0 \leq p^{max}[m,n,o] < M_b$. The value of $p^{max}[m,n,o]$ must be identified in the first packet which include B[m,n,o].

We use a second tag tree to efficiently represent $p^{max}[m,n,o]$ via the number of missing most significant planes (*Equation 6*).

(6) NUMBER OF MISSING MOST SIGNIFICANT PLANES
$q_1[m,n,o] = M_b - 1 - p^{max}[m,n,o]$

To send the relevant information, we simply invoke the tag tree coding procedure T(m,n,o,t) repeatedly for t = 1, 2, …, until $t > q_1[m,n,o]$.

$p^{max}[m,n,o]$ values do not depend upon incremental execution of the post compression rate-distortion algorithm, so there is no need to update its contents as we go.

1.2.3.3.3 Number of coding passes

For every code-block we must identify the new truncation points in terms of the total number of coding passes which will be available for decoding once the new information in this packet has been received.

Let n_i^{min} denote the lower bound on the new truncation point index (i.e. a lower bound on the number of coding passes which can be available once this packet is received).

a) IF THE CODE-BLOCK HAS ALREADY BEEN INCLUDED IN A PREVIOUS BIT-STREAM LAYER

n_i^{min} is simply one more than the number of coding passes which were available after the previous layer.

b) IF THIS IS THE FIRST TIME THE CODE-BLOCK IS BEING INCLUDED IN THE BIT STREAM

n_i^{min} might simply be 0, i.e., the first non-trivial truncation point.

For each code-block which is to be included in the bit-stream, we must identify the difference between the new truncation point and its minimum value, i.e. $n_i - n_i^{min}$. We send this difference by means of the following simple variable length code:

- If $n_i - n_i^{min} = 0$, we send a single '0' bit
- If $n_i - n_i^{min} = 1$, we send a '10'
- $2 \leq n_i - n_i^{min} \leq 4$, we send '11' + two bit representation of $n_i - n_i^{min} – 2$
- $5 \leq n_i - n_i^{min} \leq 35$, we send '1111' + five bit representation of $n_i - n_i^{min} – 5$
- $36 \leq n_i - n_i^{min}$, we send '111111111' + seven bit representation of $n_i - n_i^{min} – 5$

This allows for $n_i - n_i^{min}$ values up to a maximum of 163, which allows for values of M_b as large as 55, which should be more than sufficient for any foreseeable practical application.

1.2.3.3.4 Length information

In this section we describe the technique used to identify the number of new bytes being sent for each code-block included in the packet.

Two steps are performed to include this information:

a) the number of bits necessary to code the length information:
   As many “1” symbols are written to the stream as number of bits are necessary to binary code the length information. A “0” symbol puts an end to the counting.
b) length information:
   The binary representation of the length information is written.

2. Technical description

In our implementation we tried to respect as much as possible the structure of the original verification model software of the 2D encoder. However, several adaptations where required to transfer the 2D coding principles to a 3D system.

The described implementation of the CS- EBCOT ENCODER has been written in C language using Microsoft Visual C++ 6.0 and has been debugged using Rational Purify and Rational Visual Quantify.

2.1 Structure of the program

2.1.1 Encoder

The encoder reads from an input image written by raw data, with the most significant bit (MSB) of every pixel first (big endian), and compresses it into an output file. bit-stream. The encoder takes care of the bit-stream truncation, based on the desired bit-rate.

A general view of the encoding system is shown in the *Figure 1*.

The encoder is described in the following C files:

Name of the file	Description
coder.c	includes the main function
arithm.c	arithmetic coder
io.c	reading and writing of files
mem.c	memory allocation and initialisation
primitiv.c	primitives used in the T1 coder
stream.c	manipulation of the generated stream
t1_coder.c	core of the T1 coder
t2_coder.c	core of the T2 coder
wave_cod.c	set of 3D wavelet transforms

2.1.1.1 coder.c

void main(int argc, char* argv[])
Main module of the coder from which the main parts are launched. First of all, the input parameters are read from the command line to initialise the coder with the InitializeEbcot function. After that, the input image is read from the referenced file with the appropriate ReadImageFromFile function, depending on the number of bits per pixel of the input image. When this is done, a 3D wavelet transform is applied on the image stored in memory. Then the Tier 1 and 2 of the CS-EBCOT algorithm are applied to generate the compressed stream. Finally, the achieved bit-rate is appended to a bitrates.txt file and a results file is generated including all the information related to the coding process.

2.1.1.2 arithm.c

The core functionality of the arithmetic coder has been written by Amir Said (amir@densis.fee.unicamp.br), member of the Faculty of Electrical Engineering of the University of Campinas (UNICAMP) in Brazil, and William A.Pearlman (pearlman@ecse.rpi.edu), affiliated with the Department of Electrical, Computer and Systems Engineering of the Rensselaer Polytechnic Institute, in the United States of America.

Nevertheless, some modifications have been included to make enable a statistical analysis of the symbol probabilities for each context and the proper use of the data structures controlled by the T1 coder.

The arithmetic coder has got a maximum accuracy of 1/4095 for the probability estimation. As soon as that accuracy is reached, the arithmetic coder is set to a lower

accuracy of 2/4096 = 1/2048 and it is increased again after each coded symbol. This process must be done to avoid overflowing in the arithmetic coder.

Only the new or modified functions of the arithmetic encoder will be commented in this document.

extern void Save_Model(Adaptive *m, Adaptive_Model *m_save)
This function saves all the information of the context models in another array by adding the values field by field.

extern void Recover_Models(Adaptive_Model* M)
Initialises the *Adaptive_Model* array with the values defined in the *init_model[]* array defined in the *arithm.h* file. These values have been determined with empiric experiments (see described).

extern void Generate_Models_File(Adaptive_Model *, char *path)
Generates a file with all the information of the context models array used by the arithmetic coder.

static void Output_Byte(Encoder *E)
This function has been modified to insert a new byte into the bit-stream structure associated with the code-block currently being processed.

1.1.3 io.c

void GenerateTestData(int* wave)
Function used to generate test data for debugging purposes.

void WriteTextImage(char *path, int *wave)
Writes the content of an array of integers into a text file. Used only for debugging purposes.

void WriteTextStream(char *path, int *wave)
Writes the content of a bit-stream structure into a text file. Used for debugging purposes.

void WriteImageToFile_X(char *path, int *image) (X = 8, 16, 32)
Writes the content of an integer array, which contains the values of the voxels of the volumetric image data to an output file. The Least Significant Byte (LSB) is written first.

void ConvertIntToChars(int integer, unsigned char *output)
Splits a 16-bit representation of an integer into two 8-bit characters.

void ConvertLongToChars(long long_num, unsigned char *bytes)
Splits a 32-bit representation of an integer into four 8-bit characters.

void WriteByteToTxt(FILE *fitxer, unsigned char num)
Writes a byte into a binary format to a text file. For debugging purposes only.

void WriteBytesToTxt(FILE *fitxer, unsigned char num, int num_bytes)
Writes the desired number of bytes in a binary format to a text file. For debugging purposes only.

void WriteT1ToFile(char *path, struct ebot_encoder *ebcot)
Generates an output file with the information generated for each code-block in the T1 coder. For debugging purposes only.

void WriteResultsFile(char originalfile, char *compressedfile, char *path, clock_t start, clock_t finish)
Generates a text file with the results of the compression process, including all information related to the settings of the coder.

int ReadParameter(int argc, char *argv[], char Symbol, int Default_Value)
Reads the input command line and returns the value associated to a parameter. If the parameter is not specified in the command line then the "Default_Value" is returned.

void ReadImageFromFile_X(char *path, int *wave, unsigned int num_pixels) (X=32, 16, 8)
Reads the content of a file and copies it into a memory array of 32 bits per position. 8, 16 or 32 bits per memory position are transferred. The Least Significant Byte (LSB) is read first.

void AppendResults(char *comp_name, char *rep_name)
Appends the bit-rate of the compressed file at the end of a report file, including information about the wavelet kernel and number of decomposition levels used. It has been especially useful for massive tests on the coder.

void FindDifferences(char file1, char file2, char*rep_name, int bpp)
Compares two files and checks for any difference between them. This function was used during debugging, to check the correctness of the lossless encoder.

void WriteBytesToFile(struct iobuf *data_file, long value, int num_bytes)
Writes to a file the indicated number of bytes saved in the 32 bits of *value*. *num_bytes* cannot be larger than 4.

void WriteHeaderToFile(struct iobuf* datafile)
Based on the original JPEG2000 standard. Generates the main header of the compressed file where all settings used by the coder are written so that the decoder can initialise them properly.

Structure of the main header

Apart from the markers defined in the JPEG2000 VM 6.0, some new markers have been added to support some extra capabilities included in the CS-EBCOT coder. Basically, the structure of the main header consists of predefined markers, which control synchronization followed by the values of the parameters necessary for the decoding process.

The structure of the main header is shown in Table 10.

Table 10: Structure of the main header in CS-EBCOT compressed files

Name	Value	Number of bytes	Type	Description
SOC	0xFF4F	2	Marker	Start of codestream
SIZ	0xFF51	2	Marker	Input volume size
Xsiz		4	Data	Input volume size in X
Ysiz		4	Data	Input volume size in Y
Zsiz		4	Data	Input volume size in Z
COD	0xFF52	2	Marker	Coding style default
Xlev		1	Data	Wavelet levels in X
Ylev		1	Data	Wavelet levels in Y
Zlev		1	Data	Wavelet levels in Z
Lcod		2	Data	Number of quality layers
Xcod		1	Data	Code-block size in X
Ycod		1	Data	Code-block size in Y
Zcod		1	Data	Code-block size in Z
Xmin		1	Data	Minimum subblock size in X
Ymin		1	Data	Minimum subblock size in Y
Zmin		1	Data	Minimum subblock size in Z
Xker		1	Data	Wavelet kernel in X
Yker		1	Data	Wavelet kernel in Y
Zker		1	Data	Wavelet kernel in Z
Ibpp		1	Data	Bit-rate of input volume
Pres		1	Data	Decimal figures in floating point
Scan		1	Data	Scanning pattern
Arit		1	Data	Arithmetic coder mode
SOD	0xFF93	2	Marker	Start of data

void WriteDataToFile(struct _iobuf* fitxer, struct stream* s)
Appends the content of a stream data structure to a file already opened.

void WriteStreamToFile(char *path)
This function opens the file where the compressed stream is going to be written to. First, the main header is written, and later the data stream generated by the EBCOT encoder.

2.1.1.4 mem

int is_power_of_2(int val) [original from BARBARIAN coder]
Return 1 if *val* is a power of 2.

int GetPower2(int num)
Returns the next lower power of 2 from *num*

void AllocateBlock(strut ebcot_block_info *block, struct ebcot_blok_info *tables)
Sets all the positions of the arrays pointed by *tables* to 0 and modifies some pointers from the *block* structure to point to these *tables* arrays. Doing that reduces the memory requirements.

void DeleteBlock(struct ebcot_block_info* block)
Frees the memory allocated where some pointers of the *block* structure are referring to

void EstimateRoomInLevels(int *boundary, int lev_idx)
Calculates how many code-blocks fit in each subband. If the lowest resolution level has not been reached and the size of the code-blocks is small enough to decompose further, the LLL band of the present resolution level is decomposed again to generate a new resolution level. The same *EstimateRoomInLevels* is called recurrently.

void AllocateLevels(void)
Uses the information provided by the *EstimateRoomInLevels* function and allocates all memory necessary for the T1 coder engine. This memory is allocated according to the memory data structure explained later.

void DeleteLevels(void)
Frees the memory space allocated by the *AllocateLevels* function.

void initialize_MSE_luts(void) [Original JPEG2000 function]
Fills out the `ebcot_initial_MSE_lut' and `ebcot_refinement_MSE_lut' arrays and the lossless versions of these arrays, to assist the computation of the change in MSE that can be attributed to the new information, which is encoded in any block coding pass. Both LUT's take MSE_LUT_BITS indices, where the most significant bit of the index corresponds to the bit-plane for which the information is being coded. In the case of the `ebcot_initial_MSE_lut', the most significant bit of the index must always be 1, because the sample has just been found to be significant in the current bit-plane; thus, half the table is actually redundant, but this regular organization improves the readability of the implementation of the block coding algorithm. MSE changes are normalized so that a value of 2^13 corresponds to D^2, where D is the step size associated with the relevant bit-plane (i.e. a change in the most significant bit of the index supplied to the LUT). The lossless variants of the LUT's are to be used only when coding the least significant bit of a lossless representation of the subband samples, i.e. in reversible systems. These LUT's are a little different because the representation levels must be the quantizer thresholds themselves here, rather than the midpoints between the thresholds.

void ReturnMortonCoords(int counter, int* coord) [Barabarin code]
Returns in Cartesian coordinates the position that is associated to the value given in *counter* following a 3D Morton scanning pattern.

void initialize_morton_lut(void)
Initialises a look-up table for all the coordinates associated to each counter value. This look-up table is used to speed up the coder by computing the Morton coordinates only once in the initialisation stage.

void ebcot_initialize_global_luts(void)
Initialises the MSE and Morton look-up tables during the initialisation stage.

void InitializeEbot(int argc, char* argv[])
Initialises the basic parameters necessary to perform the coding process with the values given in the command line, or if none is given with a default value.

2.1.1.5 primitiv.c

In the functions described in this section, the neighbouring samples of the voxel being encoded are indexed as *Figure 21* shows.

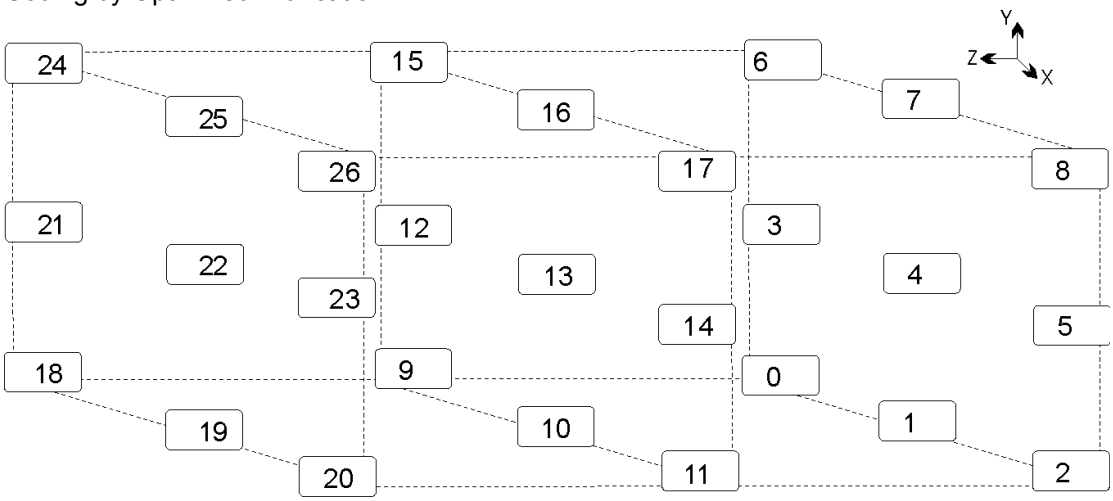


Figure 23: Indexing of neighbouring samples

char FormContextZC(int* buffer, int subband, int* coord)
Returns the context of the sample being coded in a Zero Coding primitive.

char FormContextLLL (char l, char ll, char lll)
Returns the context of the sample being coded in a Zero Coding primitive of the LLL subband depending on the number of significant samples in each of the three neighbour zones defined in the 3D mask. The context must be calculated by a function because a look-up table would be too much memory consuming. This function is also used in the HHH subband with another interpretation of the input parameters.

char FormContext_LHL(char l, char h, char ll, char hl, char lhl)
Returns the context of the sample being coded in a Zero Coding primitive of the LHL subband depending on the number of significant samples in each of the five neighbour zones defined in the 3D mask. The context must be calculated by a function because a look-up table would be too much memory consuming. This function is also used in the HLL and LLH subbands with another interpretation of the input parameters.

char FormContextHLH(char l, char h, char hh, char hl, char hhl)
Returns the context of the sample being coded in a Zero Coding primitive of the HLH subband depending on the number of significant samples in each of the five neighbour zones defined in the 3D mask. The context must be calculated by a function because a look-up table would be too much memory consuming. This function is also used in the HHL and LHH subbands with another interpretation of the input parameters.

char CheckRLC(struct ebcot_block_info *c, int x, int y, int z)
Checks if the Run Length Coding can be applied for the given coordinates.

void RunLengthCoding(struct ebcot_block_info block, int* coord, int position, int plane)
Applies the Run Length Coding primitive to a four-sample column that starts in *coord*.

char FormContextSC(int x, int y, int z)
Returns the context of the sample being coded in a Sign Coding primitive depending on the sign and significance of the surrounding samples.

void SetSignInfo(struct ebcot_block_info block, int* coord, int position)
Codes the sign information of a sample that has been found as newly significant and modifies the sign table that keeps the sign information of the significant samples.

char FormContextMR (struct ebcot_block_info c, int* coord, int position)
Returns the context of the sample being coded in a Magnitude Refinement primitive depending on the number of significant samples in each of the six closest neighbouring samples defined in the 3D mask.

2.1.1.6 stream.c

char GetNextBit(struct stream *s)
Returns the next bit of the bit-stream data structure and sets the pointers to a new position.

void SetSignInfo(struct ebcot_block_info block, int* coord, int position)
Codes the sign information of a sample that has been found as newly significant and modifies the sign table that keeps the sign information of the significant samples.

void SetLastBit(char bit, struct stream *s)
Sets the previous bit to the one pointed at that moment with the *bit*value. This is used in the octree coding pass when modifying a bit that is used to mark a non-significant cube and that it is set to significant in the present bit-plane.

void SetNextBit(char bit, struct stream* s)
Sets the next bit in the bit-stream with the value *bit* and moves the pointers to the next position.

char GetNextByte (struct stream*s)
Returns the next byte from the bit-stream.

void SetNextByte(charnum, struct stream* s)
Sets the next byte of the bit-stream with the *num* value.

void SetNextBits(struct stream* s, long num, int num_bits)
Pushes an amount of *num_bits* least-significant bits of *num* to the bit-stream, starting from the most significant of those bits.

void SetPointersToStart(struct stream *s)
Sets the pointers of a *stream* data struct to the 0 position.

void SetStreamToStart(strut stream *s)
Resets the *stream* data structure by setting the size and the pointers to zero.

2.1.1.7 t1_coder.c

int DetermineLevel (int *coord, int *boundary, int lev_idx, int *band_idx)
Returns the resolution level and the subband of a code-block whose origin is in *coord*.

void GetChildOctant(struct subblock *b)
Fills the low_*child* field of the *subblock* data structure with the origin coordinates of the child subblocks resulting from the decomposition of the subblock defined in *b*.

void AllocateCompleteMask(int *coord, int *buffer, int *contents)
Allocates the mask of 3x3 around the coord position in the buffer, inserting zeros if the mask goes further than the block's limits. Used in the Zero Coding primitive to calculate the context.

int CheckMask(int* buffer, struct subblock b)
Return 1 if there is any significant sample in the mask created around the sample that is being processed.

struct subblock CreateMask(int *coord, int *temphigh)
Return a *subblock* data structure centred in *coord* whose limits don't overcome the *temphigh* limits.

rd_slope_type convert_double_to_rd_slope_type(double slope) [Original 2D JPEG2000]
Converts a real-valued rate-distortion slope to the exponent-mantissa representation associated with the `rd_slope_type' data-type. Excessively small or large slope values are truncated to the minimum and maximum legal slopes, respectively.

void compute_rd_slopes(int num_points, double* cumulative_wmse_reduction, struct ebcot_pass_info* passes, int* rd_slope_rates) [Original 2D JPEG2000]
Computes the rate distortion slopes when coding of a code-block is finished.

int GetMSBPosition(int num)
Return the position of the Most Significant Bit in the 32-bit value *num*.

int ReadMSELut(struct ebot_block_info* block, int value)
Returns the appropriate index to read form the MSE table in order to perform the algorithm that optimises the truncation points.

int GetFirstPlane (struct ebcot_block_info *block)
Return the highest bit-plane necessary to code all the samples contained in the code-block being coded.

int CheckBlock(int* buffer, struct subblock b, int plane)
Returns 1 if the subblock defined by *b* contains at least one sample that is significant at the given bit-plane.

void RefinementPass(struct ebcot_block_info *block, struct subblock b, int plane, int coord[3], int position)
Coding pass that refines those samples that have become significant in previous bit-planes.

void SignificancePass(struct ebcot_block_info *block, struct subblok b, int plane, int coord[3], int temphigh[3], int position)
Coding pass that codes at the present bit-plane all previous non-significant samples with a preferred neighbourhood (at least one of its 26 neighbours is already significant).

void NormalizationPass(struct ebcot_block_info *blok, struct subblok b, int plane, int coord[3], int position)
Coding pass that codes at the present bit-plane all the new significant samples in the present bit-plane that were not coded in the significance.

void ScanningMorton(struct ebcot_block_info *block, struct subblock, int plane, int code)
Performs one coding pass with the 3D Morton scanning pattern -.

void Scanning2DJPEG2000(strut ebcot_block_info *block, struct subblok b, int plane, int code)
Performs one coding pass with the classical 2D JPEG2000 (stripe-wise) slice-by-slice scanning pattern code-block.

void InverseOctree (struct ebcot_blok_info *block, struct subblok b, int plane, int code)
Recovers the already coded octree information in order to code only the significant subblocks.

void CodeOctree(strut ebcot_block_info *block, struct subblok b, int plane)
Generates the octree coding information for the present bit-plane, by looking for new significant subblocks in those that were non-significant.

void CodePass(strut ebcot_block_info *block, struct subblock sb, int plane, int code_pass, int pass_idx)
Performs one coding pass of the tier 1 by applying the appropriate coding pass to the subblocks, which contains significant samples according to the octree coded stream.

void Code-block(struct ebcot_block_info block, struct ebcot_packet_band_info band)
Generates the coded bit-stream from a code-block defined in *block*.

void CodeT1(int wave, char* path)
Splits the wavelet transformed volumetric image into code-blocks and codes them using the JPEG2000 principles.

2.1.1.8 t2_coder.c

Void tag_tree__set_value(struct tag_tree_node *leaf, int value) [Original JPEG2000 function]
Sets the value of the supplied leaf node and propagates the information up the tree to higher-level nodes, which hold the minimum of their descendant node values. Must be called separately for each leaf node. It is not strictly necessary to set the value for all leaf nodes, since the initial state of the tree (after reset) is such as to ensure that all nodes have the maximum possible value. Generally speaking one must be careful about coding leaves before setting values, however, the emitted code will be properly decodable provided any encoding steps work with thresholds which are no larger than any new value which is later set into a leaf node. This property of the tag tree is exploited in the generation of inclusion masks for blocks within each subband.

void tag_tree__reset(struct tag_tree_node *tree)
Resets all values and lower bounds for the supplied tree.

void tag_tree__copy(struct tag_tree_node* src, struct tag_tree_node* dest) [Original JPEG2000 function]
Copies values and bounds from the ‘src’ tree to the ‘dest’ tree.

static int tag_tree__encode(tag_tree_node_ptr leaf, int threshold, struct stream* tags) [Original JPEG2000 function]
Encodes whether or not the ‘leaf’ node’s value is less than the supplied threshold. In the process, values that are less than the threshold are explicitly coded. Returns 1 if the value is less than the threshold and 0 otherwise. Encoded bits are pushed into the ‘target’ tag stream manager, which collects the tag codes into bytes for outputting as a single block of tags later on.

long form_packet(int rd_threshold, int simulation, int layer_idx, struct stream* tags, int lev_idx)
This function forms a new layer in the bit-stream, composed of all coding passes from all code-blocks which fall within the scope of the supplied packet, whose 'rd_slope' value is greater than or equal to the 'rd_threshold' value, and which have not already been included in a previous layer. From the perspective of the 'stream' object, the entity formed here is known as a "packet". If 'simulation' is zero, the bit-stream formed in this way will be immediately pushed out to the supplied 'stream' object. Otherwise, if 'simulation' is non-zero, nothing will be output, and the packet formation process will be simulated. In either event, the 'stream' object is used to determine the actual size of the packet 'layer_idx' identifies the layer within the bit-stream for which a packet is being formed here. The first layer should always have a 'layer_idx' value of 0, which causes appropriate initialization steps to be applied. Thereafter, the indices must increase consecutively. The 'layer_idx' value is used with each band-tile's inclusion tag tree in an interesting and elegant manner to implement an efficient coding of the point at which information from any given block is first included in the bit-stream. The 'tags' argument supplies a 'tag_buffer' object, which is used to manage the buffering of tag bits for the packet head, during the packet formation process. The function returns the actual number of bytes that are required to represent the packet, regardless of whether the 'simulation' flag is zero or non-zero. Simulation by the 'stream_out' object is required to be accurate to the byte! Failure to achieve this may result in a cascade of problems.

int optimize_bitstream _layer(int layer_idx, int max_rd_threshold, struct stream *tags, int max_cumulative_bytes, int previous_cumulative_bytes) [Original JPEG2000 function]
This function implements the rate-distortion optimisation algorithm. It saves the state of any previously generated quality layers and then invokes `form_bit-stream_layer' as often as necessary to find the smallest rate-distortion threshold such that the total number of bytes required to represent the layer does not exceed `max_cumulative_bytes' minus `previous_cumulative_bytes'. It then restores the state of any previously generated bit-stream layers and returns the threshold. The caller must invoke `form_packet' directly with this threshold to actually generate the packets that constitute the new bit-stream layer. In the extreme case, this function can be used to generate all bit-stream layers. Normally, however, it will be used only to generate a small number of layers, which are of special interest, while any intervening layer can be generated directly from some approximate threshold values. The `stream' object must be supplied, because the actual size of a packet can only be reliably computed with the aid of its interface functions, since additional error correction and/or detection codes may be added at the bit-stream level.

int estimate_layer_threshold(int target_bytes, int* rd_slope_rates, ebcot_layer_info* last_layer) [Original JPEG2000 function]
This function attempts to estimate a rate-distortion threshold that will achieve a target number of code bytes close to the supplied `target_bytes' value. It uses information of two types in order to do this. First, the `rd_slope_rates' array contains summary information collected while encoding the code-blocks (see definition of `ebcot_encoder' in "ebcot_encoder.h" for more information on this). Secondly, the `last_layer' structure contains the rate-distortion threshold and the actual number of bytes written to the bit-stream up to and including the last generated bit-stream layer.

tag_tree_node_ptr ebcot_create_tag_tree(int x_leaf, int y_leaf, int z_leaf)
Creates a tag-tree with an array of `x_leaf'*`y_leaf'*'z_leaf' leaf nodes, returning a pointer to the top-left hand corner leaf node (the leaf nodes appear at the head of the returned array, in scan-line order).

The implementations of the tag tree encoder and decoder are quite similar to their 2D versions.

A tag tree structure is defined for every subband structure. Thus, every subband structure includes as many pointers to tag tree structures as necessary.

It must be taken into account that during the packet size optimisation in the T2 coder, the tag tree structure is modified because of a simulation process. This enforces the addition of extra pointers to the tag trees to enable backup copies of the whole structures

Estimation of the memory requirements

The *subband* data structure includes all the required information to allocate the necessary memory .

Every *subband* data structure includes the sub band's dimension in code-block units. As each tag tree node is associated to a code-block, the amount of leaf nodes can be calculated dividing each dimension size by the size of the code-block in that dimension and multiplying the respective results. To calculate the amount of nodes at the level above, the obtained result has to be divided by eight (dyadic reduction of the number of nodes in each direction). The process is repeated until the root node is reached (tag tree root). The sum of the number of nodes for each tag tree level determines the total amount of memory needed to store the whole tag tree structure. The process results into a memory organisation as shown in *Figure 22*.

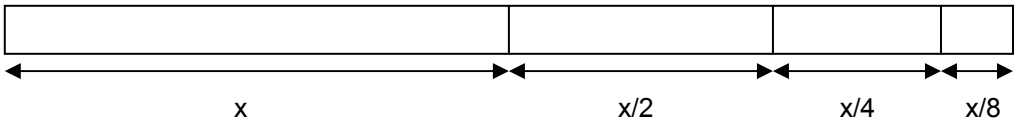


Figure 22: Memory organization of tag tree nodes

Pointers between parents and children

Each node structure includes a pointer to its child node (if the current nod is not a leaf node) and another one to his parent node (if the current node is not a root node). The end leaves nodes will have their child node pointers set to a NULL value and the root node is going to have its parent pointer set to a NULL value.

Exploiting this relation and taking into account the special characteristics of the root and the leaves, it is possible to move through the tag tree.

void CodeT2(char* path)
Manipulates the different bit-streams, generated for each code-block in the Tier 1, to create an output file defined in *path,* finding the optimal truncation points to minimize the distortion in the decompressed image.

2.1.1.9 wave_cod.c

These functions implement the three-dimensional wavelet transform of a volumetric image. It is based on a set of simple wavelet kernels that transform a one-dimensional array of samples. The three dimensional transform is achieved by calling one of these kernels for a each of the three dimensions.

Basically, these functions have been written by earlier at the ETRO department, so they are not part of this thesis. Only small modifications have been necessary to include them in front-end of our coder.

void OneDimWaveletFilter(int *InputBuffer, int *OutputBuffer, int Boundary, int Kernel)
This function performs a one-dimensional wavelet transform using kernel *Kernel*, while foreseeing a buffer boundary *Boundary* to handle the image edges and takes as data I/O *InputBuffer* and *Outputbuffer.*

void OneDimIntegerKernel5_3(int *InputBuffer, int *OutputBuffer, int Boundary)
Implementation of the 1D wavelet kernel resulting from the work of Ali Bilgin, George Zweig, Michael Marcellin "Three-dimensional image compression with integer wavelet transforms", Applied Optics, Vol.39, No.11, pp.1799-1814, 10 April 2000

void OneDimIntegerKernel_Haar(int *InputBuffer, int *OutputBuffer, int Boundary)
Implementation of the 1D wavelet kernel resulting from the work of E. H. Adelson, E. Simoncelli, and R. Hingorani, "Orthogonal pyramid transforms for image coding," in Visual Communications and Image Processing II, T. R. Hsing, ed., Proc. SPIE 845, pp.50-58 (1987).

void OneDimIntegerKernel9_7(int *InputBuffer, int *OutputBuffer, int Boundary)
Implementation of the 1D wavelet kernel resulting from the work of Ali Bilgin, George Zweig, Michael Marcellin "Three-dimensional image compression with integer wavelet transforms", Applied Optics, Vol.39, No.11, pp.1799-1814, 10 April 2000

void OneDimIntegerKernel9_3(int *InputBuffer, int *OutputBuffer, int Boundary)
Implementation of the 1D wavelet kernel resulting from the work of Ali Bilgin, George Zweig, Michael Marcellin "Three-dimensional image compression with integer wavelet transforms", Applied Optics, Vol.39, No.11, pp.1799-1814, 10 April 2000

void OneDimIntegerKernel13_11(int *InputBuffer, int *OutputBuffer, int Boundary)
Implementation of the 1D wavelet kernel resulting from the work of Ali Bilgin, George Zweig, Michael Marcellin "Three-dimensional image compression with integer wavelet transforms", Applied Optics, Vol.39, No.11, pp.1799-1814, 10 April 2000

void OneDimIntegerKernel15_11(int *InputBuffer, int *OutputBuffer, int Boundary)
Implementation of the 1D wavelet kernel resulting from the work of Ali Bilgin, George Zweig, Michael Marcellin "Three-dimensional image compression with integer wavelet transforms", Applied Optics, Vol.39, No.11, pp.1799-1814, 10 April 2000

void OneDimIntegerKernel2_6(int *InputBuffer, int *OutputBuffer, int Boundary)
Implementation of the 1D wavelet kernel resulting from the work of S. Dewitte and J. Cornelis, "Lossless integer wavelet transform," IEEE Signal Process. Lett. 4, 158-160 (1997).

void OneDimIntegerKernel_SplusP(int *InputBuffer, int *OutputBuffer, int Boundary)
Implementation of the 1D wavelet kernel resulting from the work of A. Said and W. Pearlman, "An image multiresolution representation for lossless and lossy compression," IEEE Trans. Image Process. 5, 1303-1310 (1996).

void OneDimIntegerKernel_CRF13_7(int *InputBuffer, int *OutputBuffer, int Boundary)
Implementation of the 1D wavelet kernel defined in the norm ISO/IEC JTC1/SC29/WG1 WG1N1684.

void OneDimFloatingKernel9_7(int *InputBuffer, int *OutputBuffer, int Boundary)
Implementation of the 1D floating-point wavelet kernel.

void OneLevelWaveletTransform(int BoundaryX, int BoundaryY, int BoundaryZ, int KernelX, int KernelY, int KernelZ)
Performs a one level wavelet transform in each of the three spatial dimensions using the specified kernels for each spatial orientation.

void HigherPrecision(int Precision)
Shifts up all samples of the image *Precision* positions to perform a floating point transform.

void LowerPrecision(int Precision)
Shifts down all samples of the image *Precision* positions after performing a floating-point inverse transform.

void MultiLevelWaveletTransform(int* Levels, char* Kernel, int Precision)
Performs a multilevel wavelet transform on a volumetric image.

2.1.2 Decoder

The decoder reads the compressed file generated by the coder and decompresses it, to reconstruct a lossy or lossless version of the original image.

The data flow in the decoder is the inverse of the encoder's data flow shown in *Figure .1*

Name of the file	Description
decoder.c	includes the main function
arithm.c	arithmetic decoder
io.c	file reading and writing
mem.c	memory allocation and initialisation
primitiv.c	primitives used in the T1 decoder
stream.c	manipulation of the generated stream
t1_decod.c	core of the T1 decoder
t2_decod.c	core of the T2 decoder
wave_cod.c	set of 3D inverse wavelet transforms

2.1.2.1 decoder.c

void main(int argc, char* argv[])
Main decoder module from which the other modules are launched. First of all, the decoding parameters are read from the header of the compressed file. With this information, the data structures can be allocated in memory. The Tier 2 reads from the compressed file the compressed streams for each code-block. In the Tier 1, each code-block is decoded separately to recover a lossy or lossless version of the wavelet transformed volumetric image. Finally, a three-dimensional inverse wavelet transform is performed to generate a volumetric image. This resulting image is written to the output file, and a report file is generated comparing the input and output images.

2.1.2.2 arithm.c

The core functionality of the arithmetic coder has been written by Amir Said (amir@densis.fee.unicamp.br), member of the Faculty of Electrical Engineering of the University of Campinas (UNICAMP) in Brazil, and William A.Pearlman (pearlman@ecse.rpi.edu), affiliated with the Department of Electrical, Computer and Systems Engineering of the Rensselaer Polytechnic Institute, in the United States of America.

Nevertheless, some modifications have been included to make enable a statistical analysis of the symbol probabilities for each context and the proper use of the data structures controlled by the T1 coder.

The arithmetic coder has got a maximum accuracy of 1/4095 for the probability estimation. As soon as that accuracy is reached, the arithmetic coder is set to a lower accuracy of 2/4096 = 1/2048 and it is increased again after each coded symbol. This process must be done to avoid overflowing in the arithmetic coder.

Only the new or modified functions of the arithmetic encoder will be commented in this document.

extern void Recover_Models(Adaptive_Model* M)
Initialises the array of *Adaptive_Model* with the values defined in the *init_model[]* array in the *arithm.h* file. These values have been set based on heuristic experiments described later.

static void Input_Byte(Encoder *E)
This function has been modified to read the new byte from a bit-stream structure associated to the code-block currently being processed.

2.1.2.3 io.c

void WriteTextImage(char *path, int *wave)
Writes the contents of an array of integers into a text file. Used only for debugging purposes.

void ReadStreamFromFile(struct *iobuf* data*file, struct stream* s)
Reads the content of a stream file from *path*. Used only for debugging purposes.

void ReadImageFromFile_X(char *path, int *wave, unsigned int num_pixels) (X=32, 16, 8)
Reads the content of a file and copies it into a memory array of 32 bits per position. 8, 16 or 32 bits per memory position are transferred.

void WriteImageToFile_X(char *path, int *image) (X = 8, 16, 32)
Writes the contents of an integer array that contains the values of the voxels of the volumetric image data to an output file. The Least Significant Byte (LSB) is written first.

long ConvertCharsToLongInt(unsigned char *length_bytes)
Merges four 8-bit characters into a 32-bit representation of a long integer.

void ConvertLongIntToChars(long_num, unsigned char *length_bytes)
Splits a 32-bit representation of an integer into four 8-bit characters.

void ConvertIntToChars(int integer, unsigned char *output)
Splits a 16-bit representation of an integer into two 8-bit characters.

void WriteByteToTxt(FILE *fitxer, unsigned char num)
Writes a byte into a binary format to a text file. For debugging purposes only.

void FindDifferences(char file1, char file2, char*rep_name, int bpp)
Compares two files and checks for differences. This function was used during debugging, to check the correctness of the lossless coder.

void AppendResults(char *comp_name, char *rep_name)
Appends the bit-rate of the compressed file at the end of a report file, including information about the wavelet kernel and the number of decomposition levels. It has been especially useful for extensive coder testing.

void GenerateReport (char orig_name, char comp_name, char* recons_name, char *rep_name, int bpp)
Generates a report file that compares the original and reconstructed images giving the resulting PSNR. It also includes all information related to the coding and decoding processes and the achieved bit-rate.

void CalculateCR(char* originalfile, char* compressedfile, char* reportpath)
Generates a report file, which includes the compression ratio achieved in the coding process.

long ReadBytesFromFile(struct iobuf *data_file, int num_bytes)
Reads *num*_bytes bytes (a maximum of eight) from the *data_file* and returns them in a long integer in the same order as they were written in the file and filling the MSB first.

void CheckMarker(struct _iobuf *data_file, int model)
Checks that the next two bytes of a file are the ones defined in *model*.

void ReadMainHeader (struct iobuf* data_file)
Based on the original JPEG2000 2D standard, reads the main header of the compressed file extracting all the parameters necessary for the decoder.

2.1.2.4 mem

int is_power_of_2(int val) [original from BARBARIAN coder]
Return 1 if *val* is a power of 2.

int GetPower2(int num)
Returns the next lower power of 2 from *num*

void EstimateRoomInLevels(int *boundary, int lev_idx)
Calculates how many code-blocks fit in each subband. If the lowest resolution level has not been reached and the size of the code-blocks is small enough to decompose further, the LLL band of the present resolution level is decomposed again to generate a new resolution level. The same *EstimateRoomInLevels* is called recurrently.

void AllocateLevels(void)
Using the information provided by the *EstimateRoomInLevels* function, allocates all memory necessary for the T1 coder engine. This memory is allocated following the memory data structure explained later.

void AllocateBlock(strut ebcot_block_info *block, struct ebcot_blok_info *tables)
Sets all the positions of the arrays pointed by *tables* to 0, and modifies the some pointers from the *block* structure to point to these *tables* arrays. Doing that, a large amount of memory is saved.

void DeleteBlock(struct ebcot_block_info* block)
Frees the memory referenced by some pointers of the *block* structure.

void initialize_mse_luts(void) [Original JPEG2000 function]
Fills out the `ebcot_initial_mse_lut' and `ebcot_refinement_mse_lut' arrays and the lossless versions of these arrays, to assist in computing the change in MSE that may be attributed to new information, which is encoded in any block-coding pass. Both LUT's take MSE_LUT_BITS indices, where the most significant bit of the index corresponds to the bit-plane for which the information is being coded. In the case of the `ebcot_initial_mse_lut', the most significant bit of the index must always be 1, because the sample has just been found to be significant in the current bit-plane; thus, half the table is actually redundant, but this regular organization improves the readability of the implementation of the block coding algorithm. MSE changes are normalized so that a value of 2^13 corresponds to D^2, where D is the step size associated with the relevant bit-plane (i.e. a change in the most significant bit of the index supplied to the LUT). The lossless variants of the LUT's are to be used only when coding the least significant bit of a lossless representation of the subband samples, i.e. in reversible systems. These LUT's are a little different because the representation levels must be the quantizer thresholds themselves here, rather than the midpoints between the thresholds.

void ReturnMortonCoords(int counter, int* coord) [Joeri Barabarien's code]
Returns the position in Cartesian coordinates, which is associated to the value given in *counter* following a 3D Morton scanning pattern.

void initialize_morton_lut(void)
Initialises a look-up table with all the coordinates associated to each counter value. This look-up table is used to speed up the coder as all the computation necessary to generate the Morton coordinates are only done once in the initialisation stage.

void ebcot_initialize_global_luts(void)
Initialises the MSE and Morton look-up tables during the initialisation stage.

void InitializeEbot(int argc, char* argv[])
Initialises some basic parameters and tables necessary to perform the decoding process.

2.1.2.5 primitiv.c

char FormContextZC(int* buffer, int subband, int* coord)
Returns the context of the sample being coded in a Zero Coding primitive.

char FormContextLLL (char l, char ll, char lll)
Returns the context of the sample being coded in a Zero Coding primitive of the LLL subband depending on the number of significant samples in each of the three neighbour zones defined in the 3D mask. The context must be calculated by a function because a look-up table would be too much memory consuming. This function is also used in the HHH subband with another interpretation of the input parameters.

char FormContext_LHL(char l, char h, char ll, char hl, char lhl)
Returns the context of the sample being coded in a Zero Coding primitive of the LHL subband depending on the number of significant samples in each of the five neighbour zones defined in the 3D mask. The context must be calculated by a function because a look-up table would be too much memory consuming. This function is also used in the HLL and LLH subbands with another interpretation of the input parameters.

char FormContextHLH(char l, char h, char hh, char hl, char hhl)
Returns the context of the sample being coded in a Inverse Zero Coding primitive of the HLH subband depending on the number of significant samples in each of the five neighbour zones defined in the 3D mask. The context must be calculated by a function because a look-up table would be too much memory consuming. This function is also used in the HHL and LHH subbands with another interpretation of the input parameters.

char CheckRLC(struct ebcot_block_info *c, int x, int y, int z)
Checks if the Inverse Run-Length Coding can be applied from the given coordinates.

void RunLengthDecoding(struct ebcot_block_info block, int* coord, int position, int plane)
Applies the Inverse Run-Length Coding primitive to a four-sample column that starts in coord.

char FormContextSC(int x, int y, int z)
Returns the context of the sample being coded in a Sign Coding primitive depending on the sign and significance of the samples around.

void GetSignInfo(struct ebcot_block_info block, int* coord, int position)
Decodes the sign information of a sample that has been found as new significant and modifies the sign table, which keeps the sign information of the significant samples.

char FormContextMR (struct ebcot_block_info c, int* coord, int position)
Returns the context of the sample being coded in a Magnitude Refinement primitive depending on the number of significant samples in each of the six closest neighbouring samples defined in the 3D mask.

2.1.2.6 stream.c

void SetNextBitToStream(char bit, struct stream* s)
Sets the next bit in the bit-stream with the value *bit* and it moves the pointers to the next position.

void SetLastBitToStream(char bit, struct stream *s)
Sets the previous bit to the one pointed at that moment with the *bit* value.

char GetNextBitFromFile(struct buffer *b)
Returns the next bit of the buffer data structure, which is loaded with the last byte read from a file, and sets the pointers to a new position.

long GetNextBitsFromFile(struct buffer *b, int num_bits)
This function retrieves `num_bits' bits from the tag buffer and returns them as the least significant bits of the returned 32-bit word. The least significant bit of the returned word is the last of the `num_bits' retrieved. The `num_bits' value must not exceed 32.

char GetNextBitFromStream(struct stream *s)
Return the next bit of the stream data structure and sets the pointers to the next position.

char GetNextByteFromStream(struct stream *s)
Return the next byte of the stream data structure and sets the pointers to the next position.

void SetNextByteToStream(char num, struct stream* s)
Sets the next byte in the bit-stream with the value *num* and moves the pointers to the next position.

void SetPointersToStart(struct stream *s)
Sets the pointers of a *stream* data structure to the 0 position.

void SetStreamToStart(strut stream *s)
Resets the *stream* data structure by setting the size and the pointers to zero.

2.1.2.7 t1_decod.c

int DetermineLevel (int *coord, int *boundary, int lev_idx, int *band_idx)
Returns the resolution level and the subband of a code-block whose origin is in *coord*.

void GetChildOctant(struct subblock *b)
Fills the low_*child* field of the *subblock* data structure with the origin coordinates of the child subblocks resulting from the decomposition of the subblock defined in *b*.

void AllocateCompleteMask(int *coord, int *buffer, int *contents)
Allocates the mask of 3x3 around the *coord* position in the buffer, inserting zeros if the mask goes further than the block's limits. Used in the Zero Coding primitive to calculate the context.

int CheckMask(int* buffer, struct subblock b)
Returns 1 if there is any significant sample in the mask created around the sample that is being processed.

struct subblock CreateMask(int *coord, int *temphigh)
Returns a *subblock* data structure centred in *coord* whose limits don't overcome the *temphigh* limits.

int DecodeFirstPlane (int insignificant_msbs)
Returns the highest bit-plane necessary to decode all the samples contained by processed code-block.

void InverseRefinementPass(struct ebcot_block_info *block, struct subblock b, int plane, int coord[3], int position)
Decoding pass that refines those samples, which have become significant in previous bit-planes.

void InverseSignificancePass(struct ebcot_block_info *block, struct subblok b, int plane, int coord[3], int temphigh[3], int position)
Decoding pass that decodes, at the present bit-plane, all previous non-significant samples with a preferred neighbourhood (at least one of its 26 neighbours is already significant).

void InverseNormalizationPass(struct ebcot_block_info *block, struct subblok b, int plane, int coord[3], int position)
Decoding pass that decodes, at the present bit-plane, all the new significant samples in the present bit-plane, which were not coded in the significance pass.

void ScanningMorton(struct ebcot_block_info *block, struct subblock, int plane, int code)
Performs one decoding pass following the 3D Morton scanning pattern through the code-block.

void Scanning2DJPEG2000(strut ebcot_block_info *block, struct subblok b, int plane, int code)
Performs one decoding pass following the classical 2D scanning pattern slice by slice through the code-block.

void InverseOctree (struct ebcot_blok_info *block, struct subblock b, int plane, int code)
Recovers the already coded cube-splitting information in order to decode only the significant subblocks at the present bit-plane.

void RecoverOctree(strut ebcot_block_info *block, struct subblock b)
Generates the cube-splitting coding information for the present bit-plane looking for new significant subblocks in those that were non-significant.

void CodePass(strut ebcot_block_info *block, struct subblock sb, int plane, int code_pass, int pass_idx)
Performs one coding pass of the tier 1 by applying the appropriate coding pass to the subblocks which contain significant samples according to the CS coded stream.

void DeCode-block(struct ebcot_block_info block)
Reconstructs a lossless or lossy version of the code-block from the compressed bit-stream associated to it.

void DecodeT1(int* wave)
Splits the wavelet transformed volumetric image into code-blocks and decodes them using the JPEG2000 principles.

2.1.2.8 t2_decod.c

void tag_tree__reset(struct tag_tree_node *tree)
Resets all values and lower bounds for the supplied tree.

static int tag_tree__decode(struct tag_tree_node *leaf, int threshold) [Original JPEG2000 function]
Retrieves and decodes sufficient tag bits to determine whether or not the `leaf' node's value is less than the supplied threshold. In the process, values that are less than the threshold are completely decoded and may be retrieved later by directly accessing the `value' field of `leaf'. Values that are greater than or equal to the threshold remain unknown. The function returns 1 if the leaf's value is less than the threshold and 0 otherwise.

void recover_packet_head(int lev_idx, int layer_idx)
This function does quite a bit of the work of the function `ebcot_get_packet_head_and_body'. It recovers the header information for the packet, filling in the `new_passes' field of each relevant block, as well as the relevant entries in the `passes' array for the corresponding blocks. The caller should use this information to update the `num_passes', and `total_bytes' fields for all affected blocks and to recover the code bytes themselves. Upon entry, the initial bit of the header has already been read to determine whether or not the packet contains any information.

void ebcot_get_packet_head_and_body(int layer_idx, int lev_idx)
Reads all packets associated to a layer and a resolution level. This function redistributes portions of compressed stream read among the appropiate code-blocks.

void DecodeT2(void)
This function is responsible for reading the compressed file in an ordered manner to fill in the data structures associated to each code-block. With the information recovered from Tier 2, the Tier 1 will be able to decompressed the compressed stream.

tag_tree_node_ptr ebcot_create_tag_tree(int x_leaf, int y_leaf, int z_leaf)
Creates a tag-tree with an array of `x_leaf'*`y_leaf'*'z_leaf' leaf nodes, returning a pointer to the top-left hand corner leaf node (the leaf nodes appear at the head of the returned array, in scan-line order).

2.1.2.9 wave_dec.c

These functions implement the three-dimensional wavelet inverse transform of a volumetric image. It is based on a set of simple wavelet kernels, which transform a one-dimensional array of samples. The three dimensional inverse transform is achieved by calling one of these kernels successively for a each of the three dimensions.

Basically, these functions have been written by earlier at the ETRO department, so they are not part of this thesis. Only small modifications have been necessary to include them in front-end of our coder.

void OneDimInverseWaveletFilter(int *InputBuffer, int *OutputBuffer, int Boundary, int Kernel)
Choses among the set of available wavelet kernels to transform a set of samples contained in *InptBuffer*.

void OneDimIntegerInverseKernel5_3(int *InputBuffer, int *OutputBuffer, int Boundary)
Implementation of the 1D wavelet kernel resulting from the work of Ali Bilgin, George Zweig, Michael Marcellin "Three-dimensional image compression with integer wavelet transforms", Applied Optics, Vol.39, No.11, pp.1799-1814, 10 April 2000

void OneDimIntegerInverseKernel_Haar(int *InputBuffer, int *OutputBuffer, int Boundary)
Implementation of the 1D wavelet kernel resulting from the work of E. H. Adelson, E. Simoncelli, and R. Hingorani, "Orthogonal pyramid transforms for image coding," in Visual Communications and Image Processing II, T. R. Hsing, ed., Proc. SPIE 845, pp.50-58 (1987).

void OneDimIntegerInverseKernel9_7(int *InputBuffer, int *OutputBuffer, int Boundary)
Implementation of the 1D wavelet kernel resulting from the work of Ali Bilgin, George Zweig, Michael Marcellin "Three-dimensional image compression with integer wavelet transforms", Applied Optics, Vol.39, No.11, pp.1799-1814, 10 April 2000

void OneDimIntegerInverseKernel9_3(int *InputBuffer, int *OutputBuffer, int Boundary)
Implementation of the 1D wavelet kernel resulting from the work of Ali Bilgin, George Zweig, Michael Marcellin "Three-dimensional image compression with integer wavelet transforms", Applied Optics, Vol.39, No.11, pp.1799-1814, 10 April 2000

void OneDimIntegerInverseKernel13_11(int *InputBuffer, int *OutputBuffer, int Boundary)
Implementation of the 1D wavelet kernel resulting from the work of Ali Bilgin, George Zweig, Michael Marcellin "Three-dimensional image compression with integer wavelet transforms", Applied Optics, Vol.39, No.11, pp.1799-1814, 10 April 2000

void OneDimIntegerInverseKernel15_11(int *InputBuffer, int *OutputBuffer, int Boundary)
Implementation of the 1D wavelet kernel resulting from the work of Ali Bilgin, George Zweig, Michael Marcellin "Three-dimensional image compression with integer wavelet transforms", Applied Optics, Vol.39, No.11, pp.1799-1814, 10 April 2000

void OneDimIntegerInverseKernel2_6(int *InputBuffer, int *OutputBuffer, int Boundary)
Implementation of the 1D wavelet kernel resulting from the work of S. Dewitte and J. Cornelis, "Lossless integer wavelet transform," IEEE Signal Process. Lett. 4, 158-160 (1997).

void OneDimIntegerInverseKernel_SplusP(int *InputBuffer, int *OutputBuffer, int Boundary)
Implementation of the 1D wavelet kernel resulting from the work of A. Said and W. Pearlman, "An image multiresolution representation for lossless and lossy compression," IEEE Trans. Image Process. 5, 1303-1310 (1996).

void OneDimIntegerInverseKernel_CRF13_7(int *InputBuffer, int *OutputBuffer, int Boundary)
Implementation of the 1D wavelet kernel defined in the ISO/IEC JTC1/SC29/WG1 WG1N1684.

void OneDimFloatingInverseKernel9_7(int *InputBuffer, int *OutputBuffer, int Boundary)
Implementation of the floating-point wavelet kernel.

void OneLevelInverseWaveletTransform(int BoundaryX, int BoundaryY, int BoundaryZ, int KernelX, int KernelY, int KernelZ)
Performs a one-level inverse wavelet transform in each of the three spatial dimensions using the specified kernels.

void HigherPrecision(int Precision)
Shifts up all samples of the image *Precision* positions to perform a floating point transform.

void LowerPrecision(int Precision)
Shifts down all samples of the image *Precision* positions after performing a floating point inverse transform.

void MultiLevelInverseWaveletTransform(int* Levels, char* Kernel, int Precision)
Performs a multilevel wavelet transform on a volumetric image.

2.2. Data structs

One of the major issues was to define a hierarchical and well-designed memory structure. Identifying the links between all the data structures was critical when designing the program.

2.2.1 EBCOT structures

The general EBCOT encoder data structure and the hierarchy of the separate building components (i.e. structures) are outlined in *Figure 23*. . The data structure of the decoder is similar, though a reduced version of the encoder's one. Hence, it is not going to be commented here.

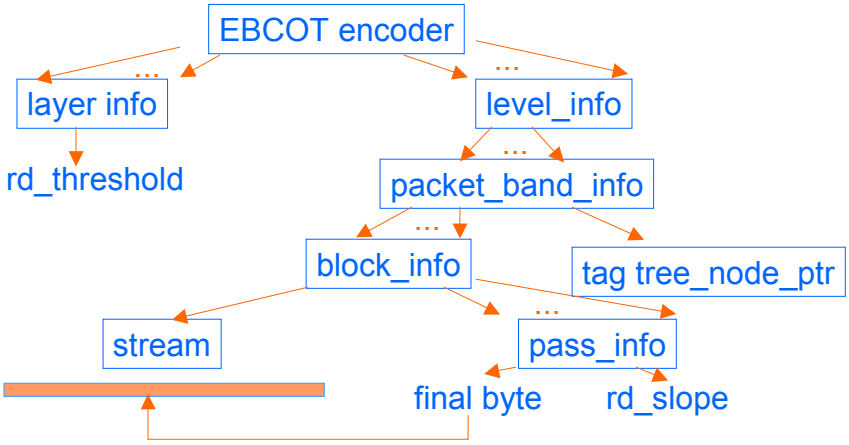


Figure 23: Main data structures

Ebcot_encoder

This structure represents the encoder object itself. It is accessible from all the program functions and acts as the root of the whole tree of data structures.

field of the data structure	description
long size_image[3]	size of the original image in the 3 dimensions
char size_block[3]	size of the code-block in the 3 dimensions
char size_min[3]	minimum size of decomposition in CS coding
char kernel[3]	code of the wavelet kernel in the 3 dimensions
int levels_wave[3]	number of levels in the wavelet transforms in the 3D
int max_level_wave	maximum level of wavelet decomposition
char precision	number of bits shifted when working with floating-point wavelet kernels
char input_bpp	bit-range of the input image
int output_bpp	Desired bit-range of the output image
int num_layers	desired number of quality layers
struct stream s	Compressed bit-stream
int* rd_slope_rates	array which contains the rate distortion slope rates
struct ebcot_layer_info* layer_info	information related to every quality layer
struct ebcot_level_info* lev	info for every resolution level
int max_lev, min_lev	maximum and minimum levels with info
int num_resol	number of resolution levels
struct _iobuf* fitxer	pointer to the compressed image file
struct _iobuf* bug_file struct _iobuf* file_t1	pointes to a bug_files (for debugging purposes only)
struct _iobuf* file_mod	pointer to a model states file for the arithmetic encoder
long symbol_count	counts the number of symbols coded
Encoder arith	structure holding the arithmetic encoder parameters
int arith_mode	flag set to 0 when starting probability 0.5, or 1 when reading probabilities from *arithm.h* file
int take_sample	flag to indicate whether a new sample for arithmetic is needed when studying context models
int num_blocks	number of coded blocks
int* image;	array containing the original image and wavelet image
int coded_bytes	number of bytes generated by arithmetic coder

Ebcot_layer_info

An instance of this data structure is defined for each quality layer. It includes the status and information necessary to determine which compressed data is going to be included.

field of the data structure	description
long max_cumulative_bytes	Maximum allowable number of bytes included in this quality layer
long actual_cumulative_bytes	number of bytes that have already been included in this quality layer
int optimize	flag indicating what function must be used to determine the rate distortion threshold
short rd_threshold	rate-distortion threshold determining the passes to be included in this quality layer

Ebcot_level_info

For each resolution level an instance of this data structure is defined.

field of the data structure	description
struct ebcot_packet_band_info bands[8]	Information related to each of the 8 subbands included in a resolution level
int room_for_blocks	Number of code-blocks that fit in this resolution level.
int min_band	First subband that includes code-blocks. (0=LLL, 7=HHH)
int max_band	Last subband that includes code-blocks. (0=LLL, 7=HHH)

Ebcot_packet_band_info

For each of the eight subbands in a resolution level, an instance of this data structure is included.

field of the data structure	description
int nr_blocks_x, nr_blocks_y, nr_blocks_z	Size of the subband counted in code-blocks in the X, Y and Z dimensions
int room_for_blocks	Number of code-blocks that fit in this subband
struct ebcot_block_info *blocks	Array of structs which includes all the information related to the code-blocks in the considered subband
tag_tree_node_ptr inclusion_tree	Tag tree structure that keeps the information related to the first inclusion layer
tag_tree_node_ptr insignificant_msbs_tree	Tag tree structure that keeps the information related to the msbs.
tag_tree_node_ptr inclusion_tree_copy	Copy of the tag tree structure that keeps the information related to the first inclusion layer
tag_tree_node_ptr insignificant_msbs_tree_copy	Copy of the tag tree structure that keeps the information related to the msbs.

Subblock

Data structure that defines a subunit of a code-block.

field of the data structure	description
int low[3]	Upper left closest coordinates of the subblock (x, y, z)
int low_child[24]	Upper left closest coordinates of the children (x, y, z)
int size[3]	Size of the subblock.

Ebcot_block_info

Includes all the information related to one code-block. .

field of the data structure	description
struct stream s	Bit-stream containing the encoded information
int max passes	Number of coding passes applied to this code-block.
int new_passes	Number of passes included in the quality level being created
int old_passes	Number of coding passes already included in previous quality layers
int new_bytes	Number of new bytes included in the quality layer being processed.
struct ebcot_pass_info passes[MAX_PASSES]	Structure containing all the information related to the coding passes applied to the code-block.
int insignificant_msbs	Number of insignificant biplanes starting at bitplane 32 and ending at the first significant bit-plane.
int *buffer	Buffer with the code-block's voxels included in this.
int *sigma	Buffer pointing to the significant samples of the code-block.
int *sign	Buffer keeping the sign information of all samples of the code-block
int *mu	Buffer pointing to the samples already visited for the present bit-plane.
double delta_MSE	Increment in the MSE.
double cumulative_wmse [MAX_PASSES]	Array keeping the weighted MSEs at the end of each coding pass.
short *initial__lut	Look-up table that helps computing the MSE in the significance and normalization passes.
short *refinement_MSE_lut	Look-up table that helps computing the MSE in the refinement pas.
struct stream octree	CS bit-stream associated to the code-block.
int subband	Subband of the code-block

Ebcot_pass_info

This structure manages the information related to the code size and the rate-distortion slope at the end of any given block coding pass.

field of the data structure	description
int final_byte	Pointer to the last bit of the coded bit-stream belonging to this coding pass.
int rd_slope	Rate-distortion slope associated to this coding pass.

Stream

This structure manages the bit-stream.

field of the data structure	description
int *data	Coded bit-stream
long size_byte	Size in bytes of the bit-stream.
long num_word	Number of complete 32-bit words in the bit-stream.
int size_bit	Number of bits filled in the word which is presently being filled.
long point_word	Pointer to the word that is currently being read.
long point_byte	Pointer to the byte that is currently being read.
int point_bit	Pointer to the bit that is currently being read in the present word.

2.2.2 Arithmetic coder structures

The program uses a context-based adaptive arithmetic coder. Consequently, a data structure array exists containing as many first-level entries as models for the context are defined.The data structure array created to manage the contexts is shown in the following T*able 11*.

Table 11: Data structure array of defined contexts

MODEL	CONTEXT	CODE	CODING OPERATION
0	0	AG	Models used to code whether Run Length Coding is possible.
1	0	ZC	Models used in the Zero Coding Operation
2	1		
3	2		
4	3		
5	4		
6	5		
7	6		
8	7		
9	8		
10	9		
11	10		
12	11		
13	12		
14	13		
15	14		
16	15		
17	0	MAG	Models used in the Magnitude Refinement Coding Operation.
18	1		
19	2		
20	0	SC	Models used in the Sign Coding Operation.
21	1		
22	2		
23	3		
24	4		
25	5		
26	0	UNI	Model used when some information is supposed to follow a uniform distribution
27	0	OCT	CS coding

2.3. Input parameters

2.3.1 Encoder

The program is command line based. For each non-specified parameter a default value is defined.

In the current implementation, 3 parameters are necessary to execute the program, the others can be omitted (in that case the default values will be taken):

Command

coder [source_image] [compressed_stream] [directory for results] [-w, -h, -d, -e, -f, …(complete list of flags shown in Table 12)]

Table 12: Flags accepted in the command line

	DESCRIPTION	Default value
w	Size of input image in the x direction	64
h	Size of input image in the y direction	64
d	Size of input image in the z direction	64
e	Size of code-blocks in the x direction	32
f	Size of code-blocks in the y direction	32
g	Size of code-blocks in the z direction	32
m	Smallest code-block decomposition on the x direction	16
n	Smallest code-block decomposition on the y direction	16
o	Smallest code-block decomposition on the z direction	16
X	wavelet kernel used in the X direction 1 (5x3), 2 (Haar), 3 (9x7), 4 (9x3), 5 (13x11), 6 (5x11), 7 (2x6), 8 (S+P), 9 (13x7), 100 (Floating 9x7) (more detail in *Table 1*)	1
Y	wavelet kernel used in the Y direction	1
Z	wavelet kernel used in the Z direction	1
x	decomposition levels of wavelet transform in X direction	5
y	decomposition levels of wavelet transform in Y direction	5
z	decomposition levels of wavelet transform in Z direction	5
p	Shift applied to samples when working with floating-point kernels	0
b	bits per voxel in the input image	8
B	maximum bit-rate of the compressed image * 10000	b
l	number of quality layers	4
c	arithmetic coder mode: 0 for starting 0.5, 1 for loading file	0
s	scanning pattern (0: 2D JPEG200 slice, 1: Morton scan)	0

2.3.2 Decoder

The decoder does not need so much information, since everything is stored in the header of the compressed file. In the decoder case, the command line should be:

Command

decoder [compressed_file] [target_file]

3. Tests

Several tests have been performed to tune the coder and compare to other state-of-art three-dimensional image coders.

3.0 Test environment

3.0.1 Set of images used for testing

A set of five volumetric medical images has been used to check the performance of the coder and its best parameters configuration. These files included only raw data with the Least Significant Byte(LSB) first. The images used are described in *Table 13*:

Table 13: Set of test images used during experiments

Code	File	Size of the file (kb)	Bits per pixel	Size (WxHxD)	File bytes per sample	Imaging device	Details
1	3DEcho	16,385	15	128x128x39	2	Ultrasound (US)	Prostate
2	3DPET	1,248	8	256x256x256	1	Positron Emission Tomogrpahy (PET)	Brain
3	Axial CT[#]	51,200	12	512x512x100	2	Computer-assisted Tomography (CT)	Axial scan of female cadaver brain (slice 100-199 of the HVP set)
4	CT Chest[##]	22,528	12	512x512x44	2	Computer-assisted Tomography (CT)	Helical scan of normal chest an mediastinum
5	MRI Brain[##]	25,600	12	256x256x200	2	Magnetic Resonance Imaging (MRI)	T1 weighted field echo 3D volume scan of normal brain

[#] Visible Human Project data set (http://www.nlm.nih.gov/research/visible/)
[##] DICOM test set

3.0.2 Set of wavelet filters used in the tests

The wavelet filters used in the experiments have been identified in the tables and graphs with the code shown in *Table 14*:

Table 14: Lossless integer lifting filters supported by the 3D WT module. The number of filter taps - l and h - for the low-pass and high-pass analysis filters respectively, is given as lxh.

Filter Name	Code	Number of Filter Taps	Lifting Steps
5x3 [1]	1	5x3	$d[n] = x[2n+1] - \lfloor 1/2\,(x[2n] + x[2n+2]) + 1/2 \rfloor$ $s[n] = x[2n] + \lfloor 1/4\,(d[n-1] + d[n]) + 1/2 \rfloor$
S [2] (Haar)	2	2x2	$d[n] = x[2n+1] - x[2n]$ $s[n] = x[2n] + \lfloor 1/2\, d[n] \rfloor$
9x7 [1]	3	9x7	$d[n] = x[2n+1] - \lfloor 9/16\,(x[2n] + x[2n+2]) - 1/16\,(x[2n-2] + x[2n+4]) + 1/2 \rfloor$ $s[n] = x[2n] + \lfloor 1/4\,(d[n-1] + d[n]) + 1/2 \rfloor$
9x3 [1]	4	9x3	$d[n] = x[2n+1] - \lfloor 1/2\,(x[2n] + x[2n+2]) + 1/2 \rfloor$ $s[n] = x[2n] + \lfloor 19/64\,(d[n-1] + d[n]) - 3/64\,(d[n-2] + d[n+1]) + 1/2 \rfloor$
13x11 [1]	5	13x11	$d[n] = x[2n+1] - \left\lfloor \begin{array}{c} 75/128\,(x[2n] + x[2n+2]) - 25/256\,(x[2n-2] + x[2n+4]) \\ + 3/256\,(x[2n-4] + x[2n+6]) + 1/2 \end{array} \right\rfloor$ $s[n] = x[2n] + \lfloor 1/4\,(d[n-1] + d[n]) + 1/2 \rfloor$
5x11 [1]	6	5x11	$d^{(1)}[n] = x[2n+1] - \lfloor 1/2\,(x[2n] + x[2n+2]) + 1/2 \rfloor$ $s[n] = x[2n] + \lfloor 1/4\,(d^{(1)}[n-1] + d^{(1)}[n]) + 1/2 \rfloor$ $d[n] = d^{(1)}[n] - \lfloor 1/16\,(-s[n-1] + s[n] + s[n+1] - s[n+2]) + 1/2 \rfloor$
2x6 [3]	7	2x6	$d^{(1)}[n] = x[2n+1] - x[2n]$ $s[n] = x[2n] + \lfloor 1/2\, d^{(1)}[n] \rfloor$ $d[n] = d^{(1)}[n] - \lfloor 1/4\,(s[n-1] - s[n+1]) + 1/2 \rfloor$
S+P [4]	8	2x6	$d^{(1)}[n] = x[2n+1] - x[2n]$ $s[n] = x[2n] + \lfloor 1/2\, d^{(1)}[n] \rfloor$ $d[n] = d^{(1)}[n] + \lfloor 2/8(s[n-1] - s[n]) + 3/8(s[n] - s[n+1]) + 2/8 d^{(1)}[n+1] + 1/2 \rfloor$
13x7 [5]	9	13x7	$d[n] = x[2n+1] - \lfloor 9/16\,(x[2n] + x[2n+2]) - 1/16\,(x[2n-2] + x[2n+4]) + 1/2 \rfloor$ $s[n] = x[2n] + \lfloor 80/256\,(d[n-1] + d[n]) - 16/256\,(d[n-2] + d[n+1]) + 1/2 \rfloor$

3.1 Evaluation of the implemented integer wavelet filter for lossless compression

TARGET
Determine the performance of the integer wavelet filters implemented in the coder for a lossless compression.

Not all possible amounts of decomposition levels have been tested, as some previous tests with these images had already been reported by Schelkens and Barbarien [Sch00a]. In most of the cases it was clear that the optimal number of decomposition levels was five. In other cases, the resolution of images limited the choice of testing on four decomposition levels (E.g. CT).

In the first tests, the kernels for the X an Y direction were identical, but we experimented with alternitive kernels in the Z direction. This was done because medical imagery usually does not produce data with the same precision in the Z direction as in the X-Y directions. However, the results showed that kernels performing optimal along the X- and –Y-axes, tended to be the optimal to for the Z-axis.Hence, it was decided to work with identical kernels in the three spatial directions.

SETTINGS FOR THE CODER
Block size (32,32,32)
Subblock minimum size (16,16,16)
Shifting in floating point 0
Number of layers 4
Arithmetic coder mode 0

RESULTS
Although it is clear that the results are data and filter kernel dependent, it is possible to draw some general conclusions. In order of performance, we can state that the 13x11, 9x7, 5x11, 13x7 and S+P give the best results, closely followed by the second group, i.e. the 5x3 and 9x3 kernels. Only the Haar kernel performed poorly.

Table 15: Best coding transform for lossless coding for a 5-level wavelet decomposition

Filter		Bitrate				
XY	Z	3DEcho	3D PET	Axial CT	CT Chest	MRI Brain
1	1	3.716886	8.611165	3.806308	4.980506	3.954777
2	2	4.095304	9.561987	4.327164	5.30925	4.453807
3	3	3.655986	8.15764	3.647849	4.954887	3.775734
4	4	3.740604	8.648663	3.84171	5.011718	3.982697
5	5	3.657062	7.965294	3.611909	4.970834	3.718029
6	6	3.66708	8.130972	3.658911	4.956915	3.790333
7	7	3.769114	8.703062	3.889699	5.064234	4.02103
8	8	3.683502	8.352614	3.710161	4.961909	3.827486
9	9	3.659272	8.178836	3.664898	4.968798	3.79532
Wavelet Levels		5	4	5	4	5

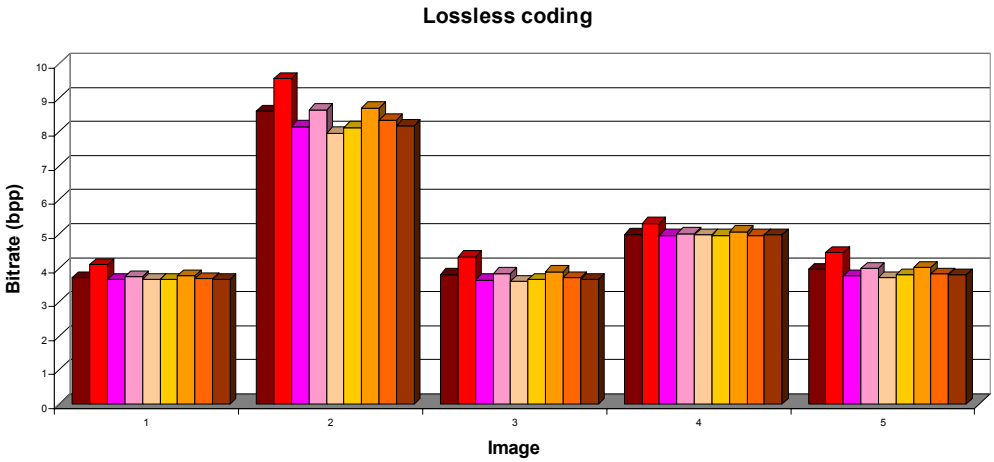


Figure 24: Best coding transform for lossless coding

3.2 Evaluation of the implemented integer wavelet filters for lossy compression

TARGET
Determine the performance of the integer wavelet filters implemented in the coder for a lossy compression.

The compression and decompression algorithms have been applied to each image for the whole set of lossless filters and for different desired bit rates. After that, a special program written in C for these tests was responsible for calculating the PSNR between the original image and the reconstructed one.

The MSE and PSNR are defined in *Equations 7* and *8*.

(7) MSE (Mean Square Error)	$$MSE = \sum_{n} \left(x[n] - y[n] \right)^2$$

(8) PSNR (Peak Signal to Noise Rate)	$$PSNR = 10 \cdot \log \left(\frac{bpp^2}{MSE / volume} \right)$$ (dB)

where *bpp* is the bit-range for the input image.

PSNR measurements were done at six different bit-rates: 2, 1, 0.5, 0.25, 0.125 and 0.0625 bits-per-pixel (bpp). However, an important consideration has to be made: the EBCOT-principles define that only some points in the coded bit-stream for each block are valid for truncation. This restriction makes it impossible to reach the desired bit-rate with an exact precision. The rate-distortion optimisation algorithm identifies the best truncation points, being as close as possible to the desired bit-rate

RESULTS
Although it is clear that the results are data and filter kernel dependent, it is possible to draw some general conclusions. At low bit rates we notice that especially the 9x3, 5x11, 9x7 and 5x3 kernels are performing well, closely followed by the 13x7 and 13x11 kernels. At higher bit-rates the 9x3, 13x11 and 5x11 filters are moving towards the other ones, which is logic since they were among the best filters for lossless coding. Notice also that the 5x11 kernel seems to be the most stable one, delivering an excellent performance over the complete lossy-to-lossless range.

3D Echo

Table 16: Best coding transform for lossy coding

		PSNR(dB)					
Filter	*Levels*	**2 bpp**	**1 bpp**	**0,5 bpp**	**0,25 bpp**	**0,125 bpp**	**0,0625 bpp**
1	5	40,32	35,81	31,60	**27,61**	**25,90**	22,06
2	5	35,43	30,58	27,22	24,42	23,62	20,47
3	5	39,46	34,95	30,89	26,90	25,18	**24,70**
4	5	**40,63**	**35,87**	**31,62**	27,56	25,82	21,95
5	5	38,96	34,46	30,60	26,56	24,69	24,27
6	5	40,17	35,48	31,28	27,16	25,62	21,53
7	5	37,34	32,05	28,31	26,35	23,42	22,95
8	5	37,15	31,8	28,22	25,95	23,22	22,73
9	5	39,58	34,72	30,52	26,65	25,14	24,67

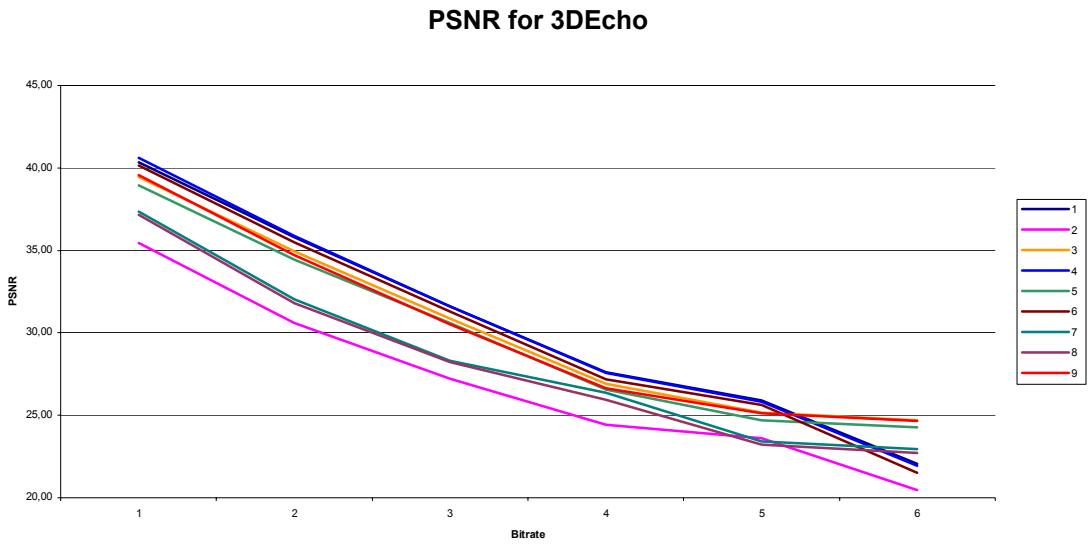


Figure 25: Best coding transform for lossy coding

3D PET

Table 17: Best coding transform for lossy coding

		PSNR(dB)					
Filter	**Levels**	**2 bpp**	**1 bpp**	**0,5 bpp**	**0,25 bpp**	**0,125 bpp**	**0,0625 bpp**
1	5	56,53	49,55	45,43	42,67	40,38	38,86
2	5	50,00	45,31	41,33	39,37	36,15	35,62
3	5	57,39	50,44	46,03	42,94	41,25	39,06
4	5	56,52	49,81	45,56	42,77	40,81	39,18
5	5	57,60	50,39	46,24	**43,45**	**41,34**	39,04
6	5	**57,97**	50,34	46,10	43,03	40,91	39,41
7	5	53,64	46,65	43,93	42,11	40,21	38,32
8	5	55,91	47,24	44,94	42,54	40,65	38,45
9	5	57,43	**50,43**	**46,38**	43,24	40,99	**39,44**

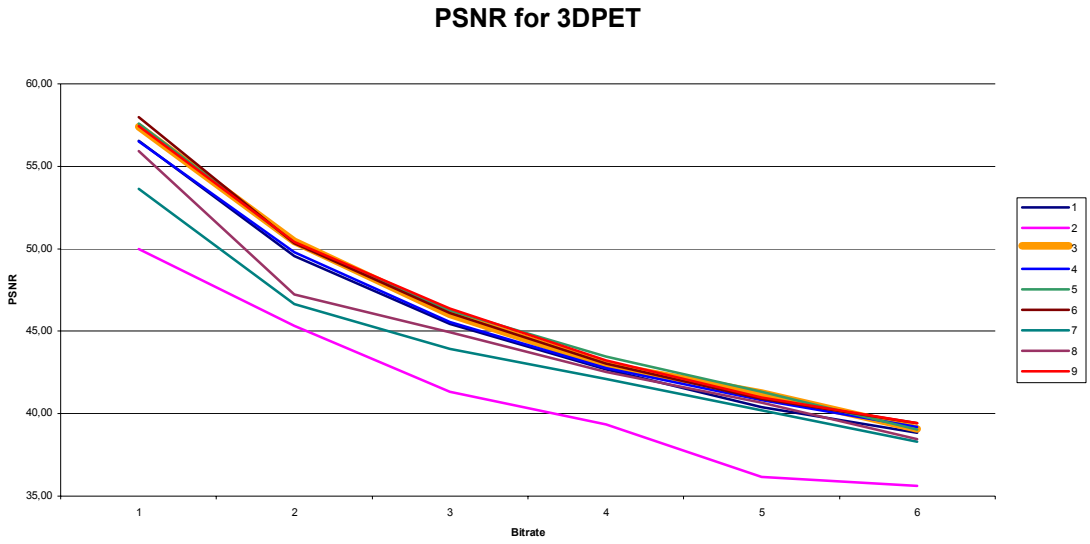


Figure 26: Best coding transform for lossy coding

Axial CT

Table 18: Best coding transform for lossy coding

		PSNR(dB)					
Filter	**Levels**	**2 bpp**	**1 bpp**	**0,5 bpp**	**0,25 bpp**	**0,125 bpp**	**0,0625 bpp**
1	5	65,57	58,90	53,93	49,76	45,44	41,14
2	5	60,98	53,86	48,41	39,73	32,86	28,58
3	5	66,35	60,03	54,79	**50,51**	46,16	41,59
4	5	65,62	59,11	54,16	49,21	45,44	41,28
5	5	**66,57**	**60,27**	**54,91**	50,31	46,22	39,56
6	5	66,43	60,04	54,59	50,49	**46,40**	**41,67**
7	5	64,26	56,95	51,92	46,86	41,56	36,83
8	5	65,34	58,91	53,70	47,10	41,51	36,66
9	5	66,43	60,18	54,82	50,33	46,37	41,30

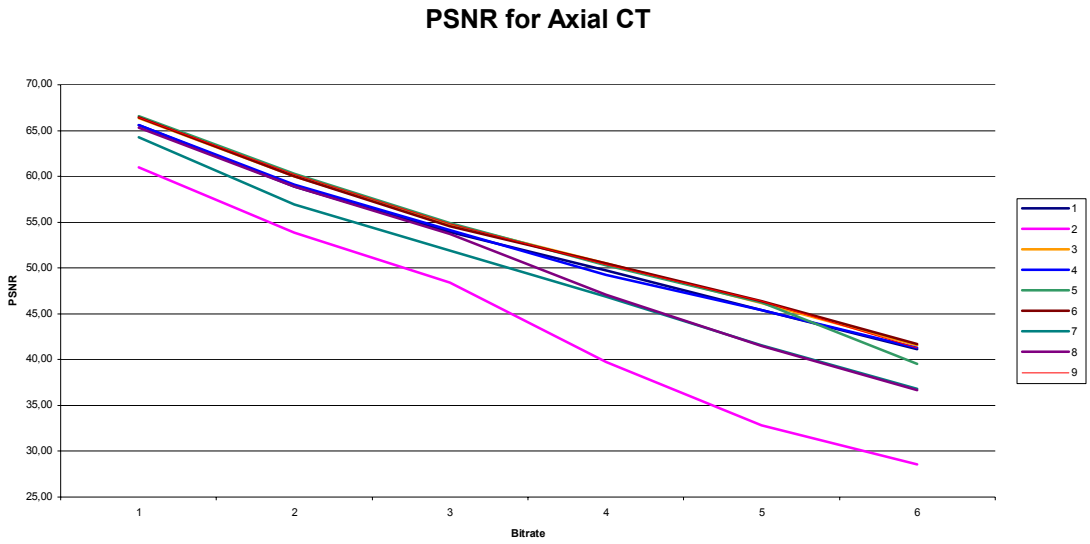


Figure 27: Best coding transform for lossy coding

CT Chest

Table 19: Best coding transform for lossy coding

		PSNR(dB)					
Filter	**Levels**	**2 bpp**	**1 bpp**	**0,5 bpp**	**0,25 bpp**	**0,125 bpp**	**0,0625 bpp**
1	5	56,36	**50,28**	**46,89**	**43,08**	39,02	34,50
2	5	52,99	46,48	42,23	37,69	33,31	28,24
3	5	56,37	50,05	46,38	42,66	38,40	33,90
4	5	56,40	50,24	46,75	43,07	**39,09**	**34,54**
5	5	56,38	49,92	46,27	42,40	38,14	33,72
6	5	**56,51**	50,25	46,59	42,95	38,90	34,43
7	5	54,96	48,20	44,37	40,15	36,03	31,85
8	5	54,59	48,23	44,09	40,08	35,81	32,19
9	5	56,41	49,97	46,44	42,61	38,25	33,72

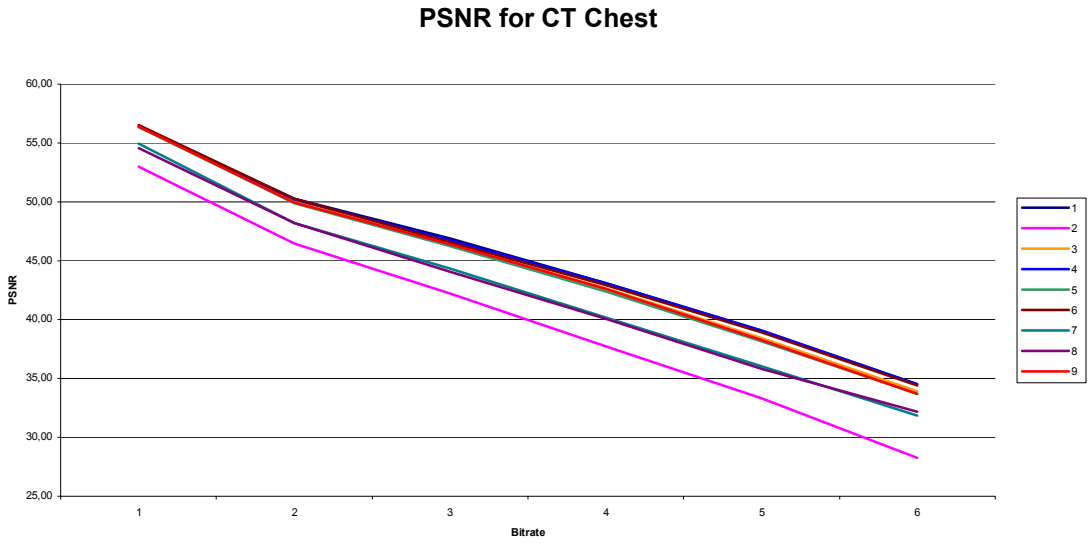


Figure 28: Best coding transform for lossy coding

MRI Brain

Table 20: Best coding transform for lossy coding

		PSNR(dB)					
Filter	**Levels**	**2 bpp**	**1 bpp**	**0,5 bpp**	**0,25 bpp**	**0,125 bpp**	**0,0625 bpp**
1	5	63,66	58,35	53,44	49,47	46,83	43,76
2	5	56,81	50,41	44,28	39,95	39,02	35,36
3	5	63,97	58,51	54,27	49,41	**47,28**	**44,05**
4	5	63,88	58,13	53,66	49,77	45,85	43,95
5	5	63,86	58,55	**54,50**	**50,68**	47,19	43,92
6	5	**64,48**	**58,82**	53,80	49,97	47,11	43,99
7	5	59,65	53,61	48,90	43,54	42,25	38,58
8	5	62,26	54,02	51,47	46,72	42,33	38,64
9	5	64,13	58,09	53,56	49,53	44,95	40,74

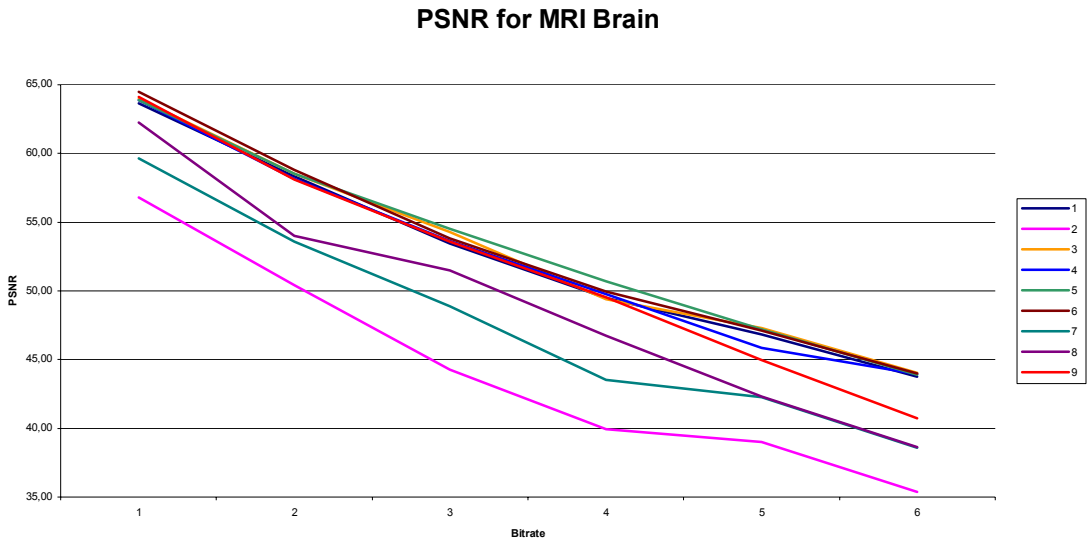


Figure 29: Best coding transform for lossy coding

3.3 Optimisation of the context classification and starting probabilities

TARGET
Study the behaviour of the models designed for the context-based adaptive arithmetic coder, in order to set a starting probability to each model. Check if there is any gain by doing this.

The arithmetic coder used in this implementation is adaptive and context-based. It is reset at the beginning of the coding of every new code block. In order to optimise its performance, several tests have been performed to find the best context definition and starting probabilities for images types for which this software is designed.

Based on the contexts defined for the JPEG2000 VM, a new set of contexts was defined for the 3D case. For the first test all probabilities were set to 0.5. As the arithmetic coder is adaptive, it will automatically adjust the initial model probabilities to more appropriate values while the image is being encoded.

As the used arithmetic coder is context-based, a different model for every context is defined. For each model separately the best starting probability has to be identified.

a) data retrieved for each image

Three properties are evaluated in every experiment:

- the probability evolution of the 0 symbol for each model during the first 1000 symbols of each pass *(prob0_x*, where x is the model code)

- number of calls to every model (*calls.txt*)
- probability of the 0 symbol for each model (*prob0.txt*)

It must be taken into account that this data was taken for each code-block, so all the data had to be averaged to have a typical response for the complete image.

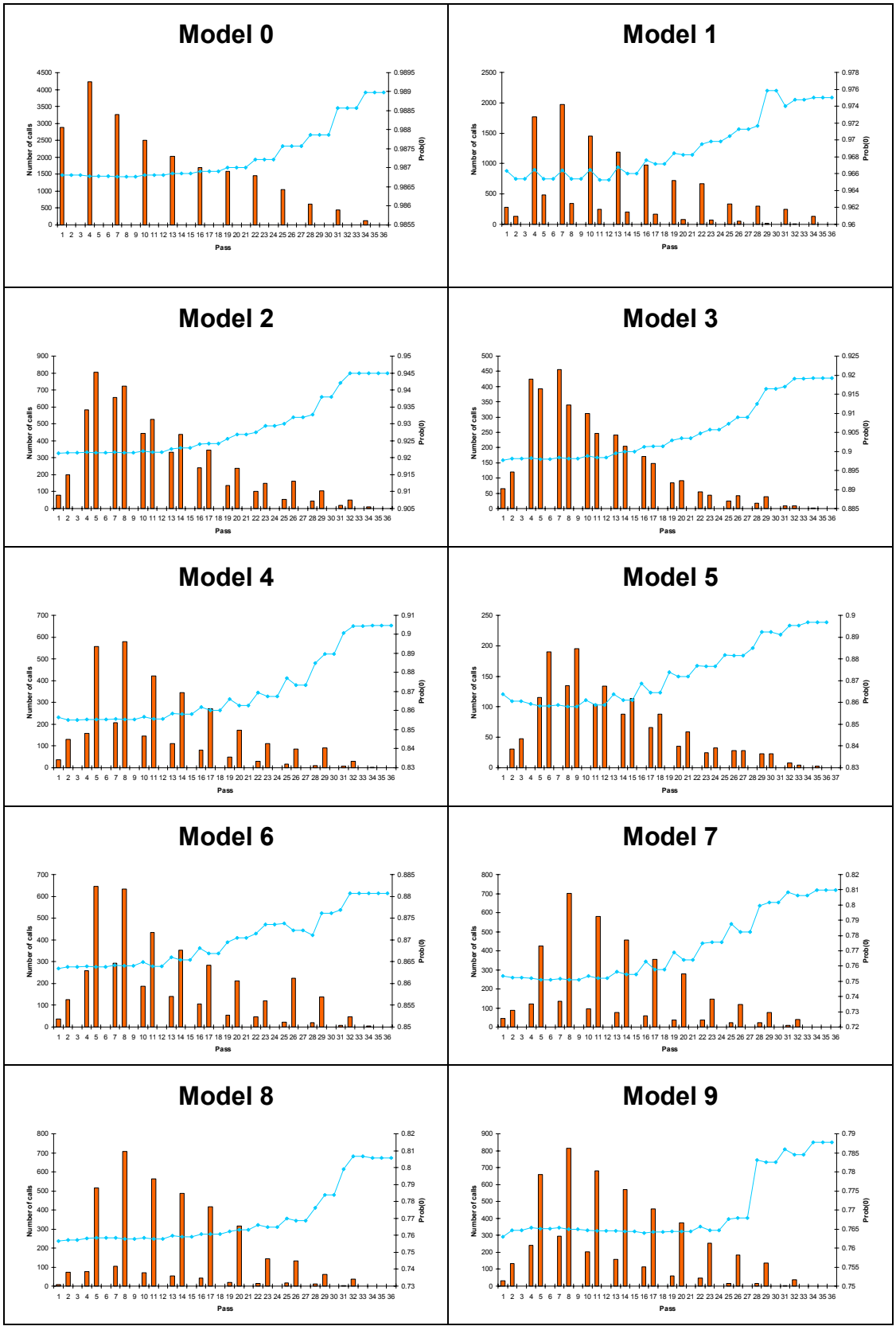
This study was performed over the whole set of images and was later averaged.

The starting probability was 0.5 .

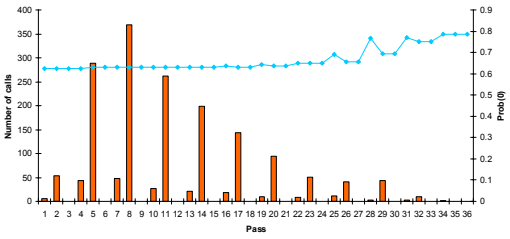
The results are commented in *Table 21*.

Table 21: Comments on behaviour of different context models defined

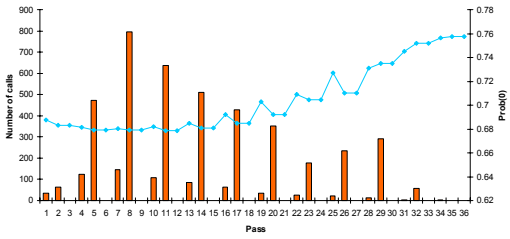
MODEL	CODE	CODING OPERATION
0	AG	Prob(0) is very high, which means that the RLC primitive, which is mainly called in the higher bit-planes, is not able to code four bits with only one symbol. However, there is some gain with it. At lower high-planes, there are less calls to the RLC with a lower chance of success.
1	ZC	The Prob(0) decreases as we move to higher contexts, which is correct as they are linked with the samples with many significant neighbouring samples. The most-called model is by far the first one, where there are no neighbouring significant samples. Most of the calls to these models are in the second or third bit-planes, where most of the samples jump from not significant to significant. At lower bit-planes there are fewer calls, so the bias shown in these bit-planes is probably due to the small amount of samples.
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		
16		
17	MAG	The first model is massively called first, then most of the refined samples jump to the second model and finally, most of the samples are refined by the third model which correspond to the case, where a sample has already been called. This is the most called model, whose Prob(0) moves around 0,5 as we are already refining samples.
18		
19		
20	SC	The Prob(0) of this group move quite close to the 0,5 probability, so no significant gain is obtained with this model. Calls to these models are only performed in the Normalization and especially Significance passes,. As these models are called every time a new sample becomes significant, we know that the significance pass is much more successful than the Normalization pass when finding new significant samples. This is because new significant samples are usually next to a previous significant sample.
21		
22		
23		
24		
25		
26	UNI	As it is a uniform model, Prob(0) really moves around the 0,5 value
27	OCT	The octree Prob(0) probability moves close to 0,56. The biggest number of calls to this model is in the first pass, when the first octree structure is created. After that, only information to refresh the octree stream is encoded, so there are not so many calls any more.


Model 0
Model 1
Model 2
Model 3
Model 4
Model 5
Model 6
Model 7
Model 8
Model 9
Number of calls
Prob(0)
Pass

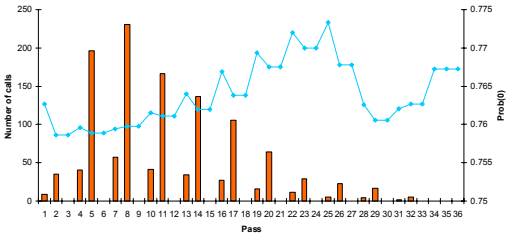
Model 10



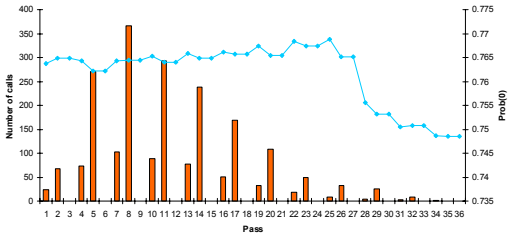
Model 11



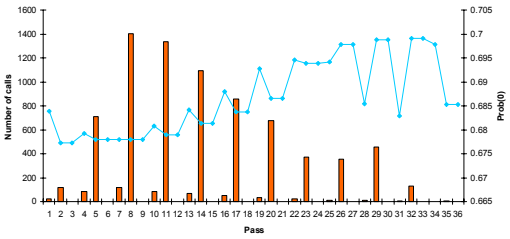
Model 12



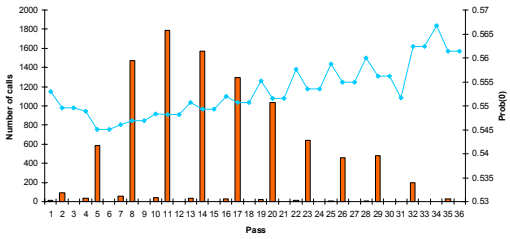
Model 13



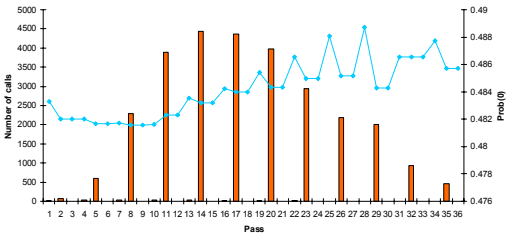
Model 14



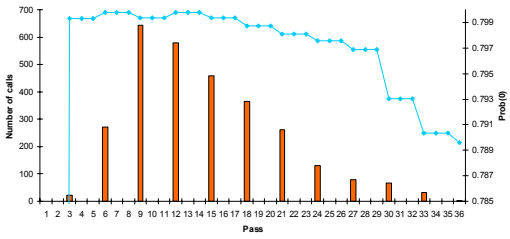
Model 15



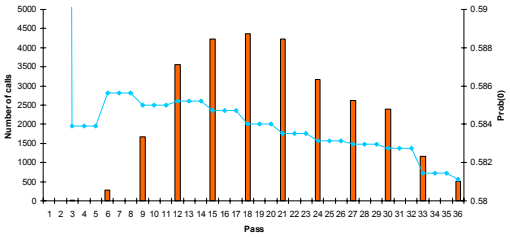
Model 16



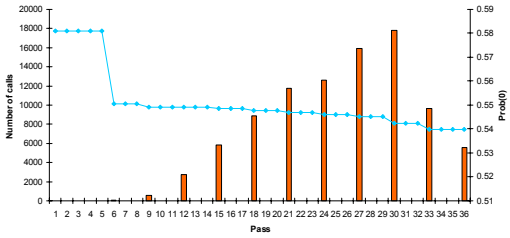
Model 17



Model 18



Model 19



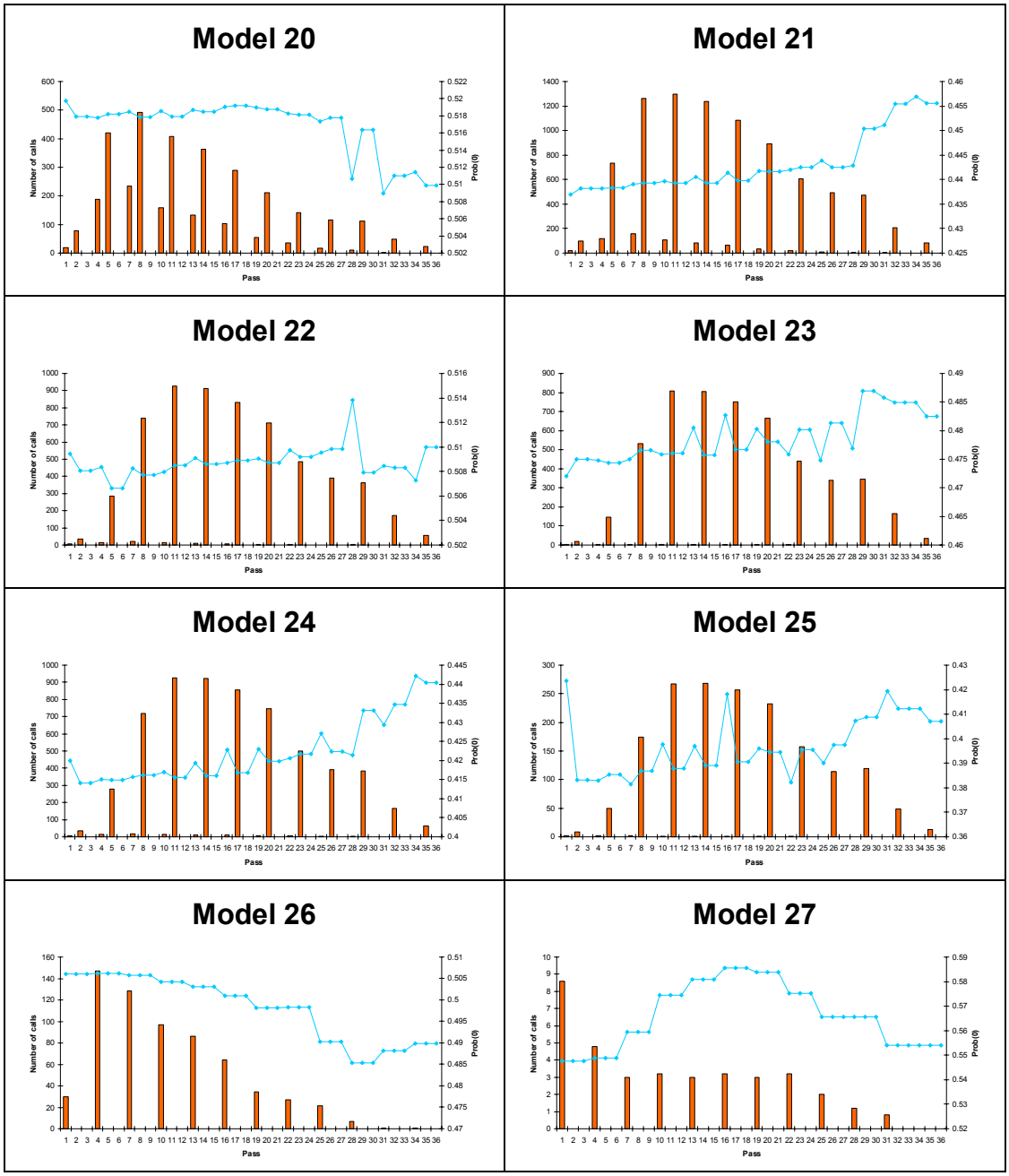


Figure 30: Numbers of calls and prob(0) evolution of each context model

b) data processing

A little program in C called *process.c* was written to perform a data analysis of the data retrieved.

With the data retrieved from each image, the three passes with the higher number of calls for each model were identified. Doing that we focus only on the passes with a higher weight on each model. This data can be found on the file *context.txt*.

The passes with more calls for each model were the following:

Table 22: Passes where the context models have been most called

Model	Most called pass	Total number of calls	Model	Most called pass	Total number of calls
0	3	21166	14	7	7007
1	6	9852	15	10	8936
2	4	4022	16	13	22150
3	6	2279	17	8	3213
4	7	2896	18	17	21874
5	7	978	19	29	89187
6	4	3225	20	7	2462
7	7	3511	21	10	6494
8	7	3530	22	10	4631
9	7	4067	23	10	4038
10	7	1845	24	10	4625
11	7	3972	25	13	1342
12	7	1154	26	3	735
13	7	1834	27	0	43

c) adapt starting probabilities

The three first decimal figures of the results were taken into account to fill the table included in *arithm.h* with the starting probabilities of each model. By doing that, the initial probabilities for the Most Probable Symbol (MPS) of each model were set to the average value of the probability of the MPS at the end of the heaviest pass:

Table 23: Probability of 0 symbol at the end of the pass with the highest number of calls for each model

Model	Start prob(MPS)	Model	Start prob(MPS)	Model	Start prob(MPS)	Model	Start prob(MPS)
0	0.986	7	0.751	14	0.677	21	0.561
1	0.966	8	0.757	15	0.548	22	0.508
2	0.921	9	0.764	16	0.517	23	0.524
3	0.898	10	0.629	17	0.799	24	0.585
4	0.855	11	0.679	18	0.584	25	0.611
5	0.858	12	0.759	19	0.542	26	0.506
6	0.863	13	0.764	20	0.517	27	0.547

The lossless and lossy tests were repeated again to test if this context optimisation was useful. The results were the following for filter number 5 and 5 decomposition levels:

Table 23b: Performance of chosen probabilities against the 0.5 probability case

	lossless (bit-rate)		2 bpp (PSNR)		0.25 bpp (PSNR)	
	start 0.5	**optimised**	**start 0.5**	**optimised**	**start 0.5**	**optimised**
3DEcho	3.657	3.665	38.96	38.93	26.56	26.53
3DPET	7.966	7.976	57.60	53.44	43.45	42.45
AxialCT	3.612	3.625	66.57	66.45	50.31	50.22
CTChest	4.971	4.985	56.38	52.93	42.40	41.37
MRIBrain	3.718	3.733	63.86	63.82	50.68	50.64
	0.125 bpp (PSNR)		**0.0625 bpp (PSNR)**			
	start 0.5	**optimised**	**start 0.5**	**optimised**		
3DEcho	24.69	24.68	24.27	24.27		
3DPET	41.34	39.74	39.04	38.62		
AxialCT	46.22	46.17	39.56	39.48		
CTChest	38.14	37.06	33.72	32.29		
MRIBrain	47.19	47.18	43.92	43.92		

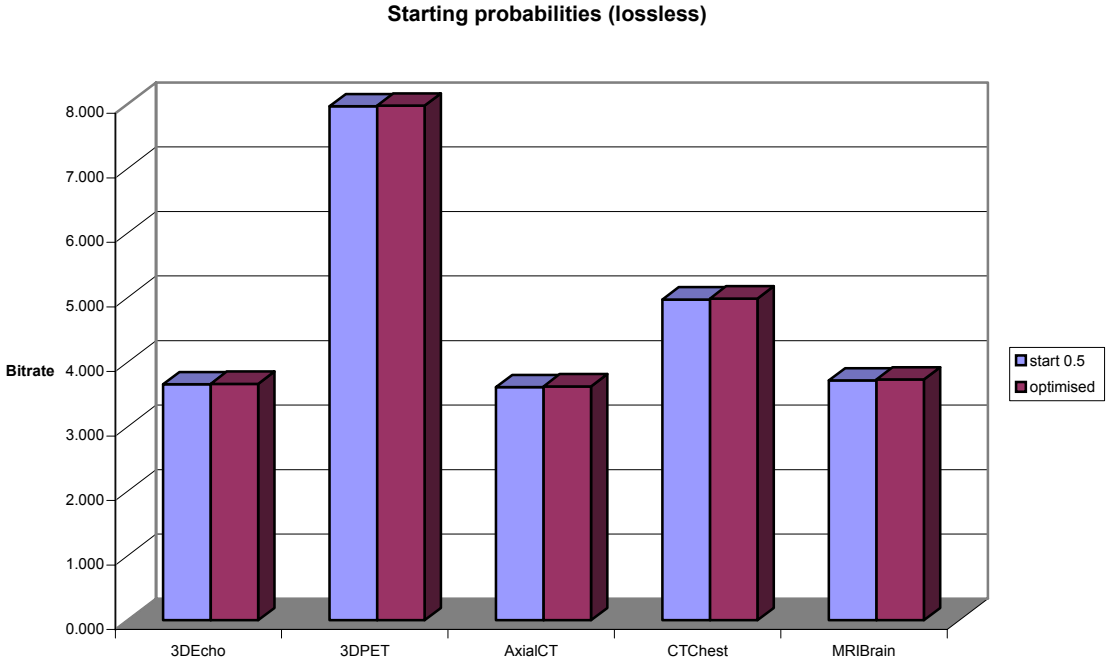


Figure 31: Performance of chosen probabilities against the 0.5 probability case

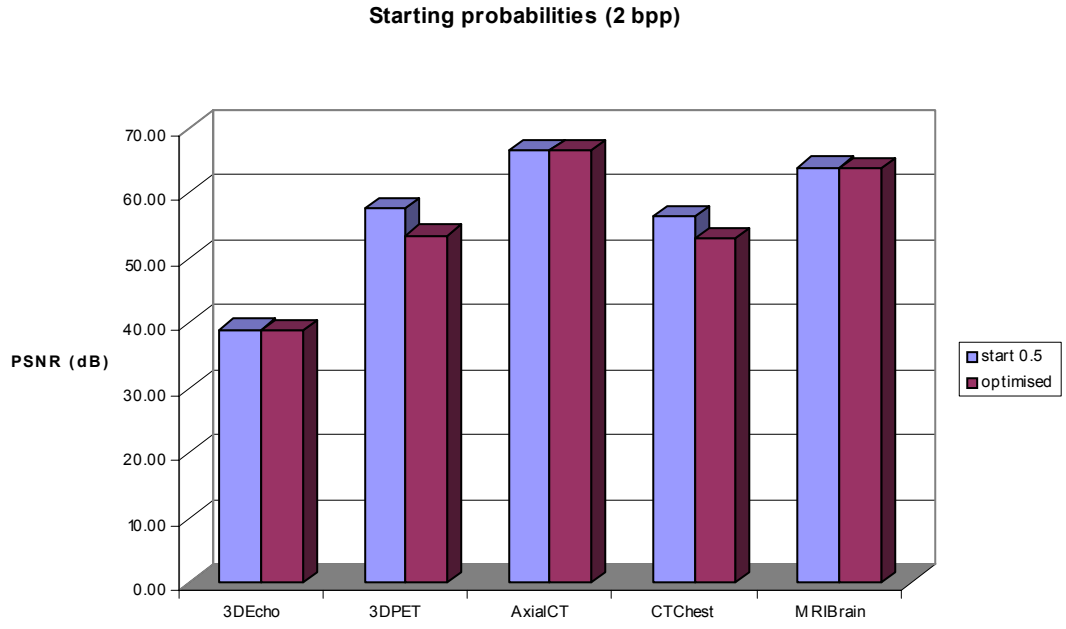


Figure 32: Performance of chosen probabilities against the 0.5 probability case

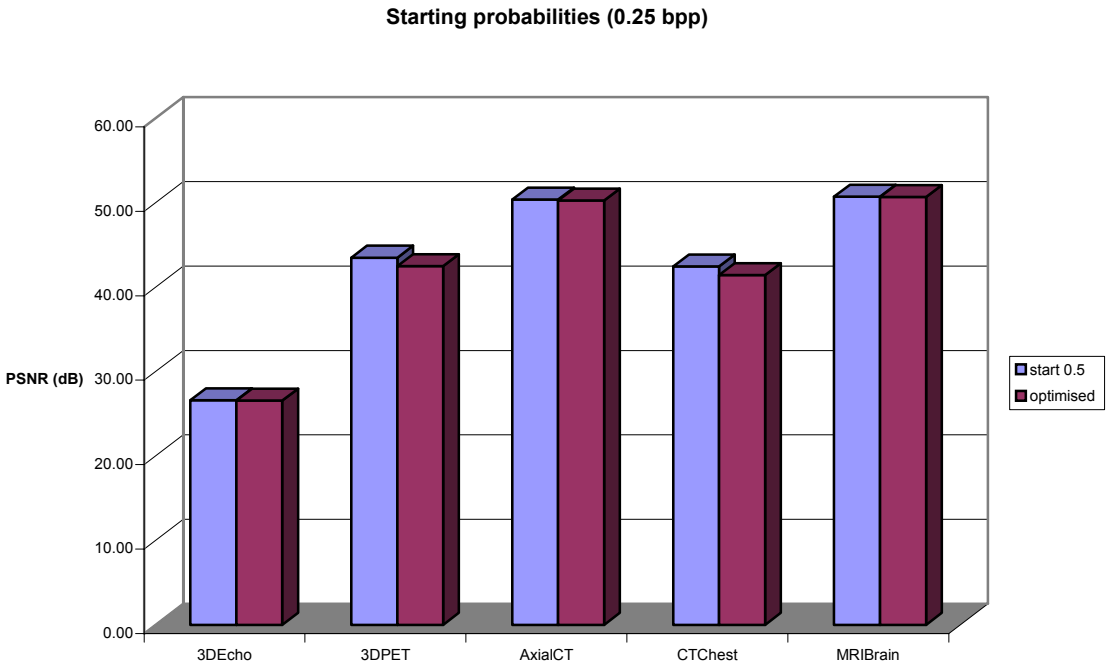


Figure 33: Performance of chosen probabilities against the 0.5 probability case

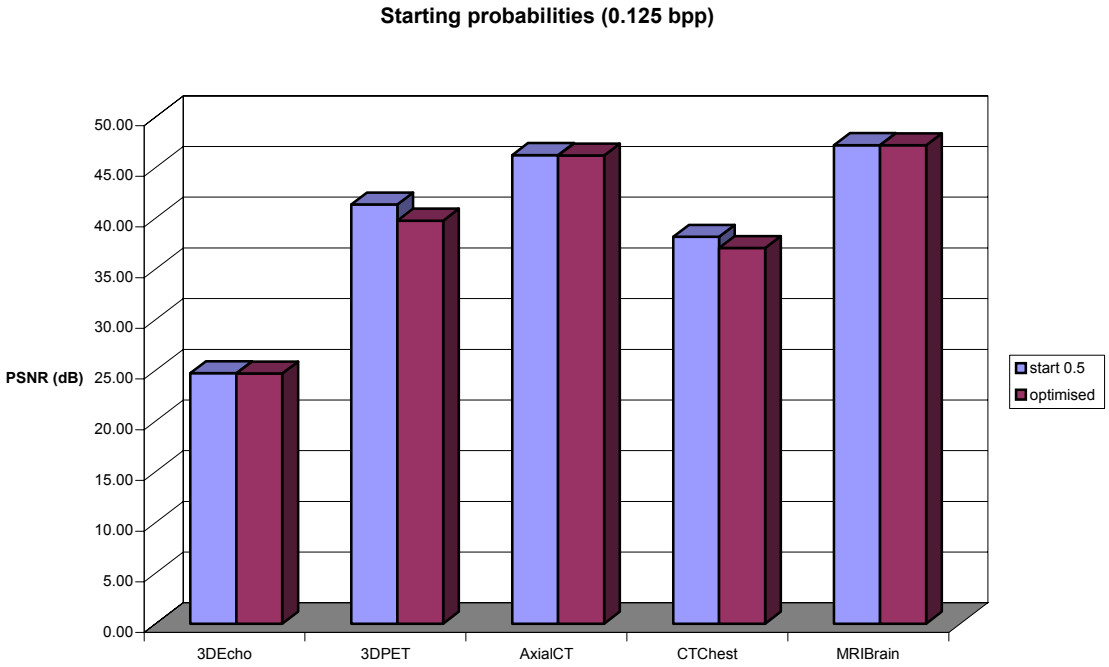


Figure 34: Performance of chosen probabilities against the 0.5 probability case

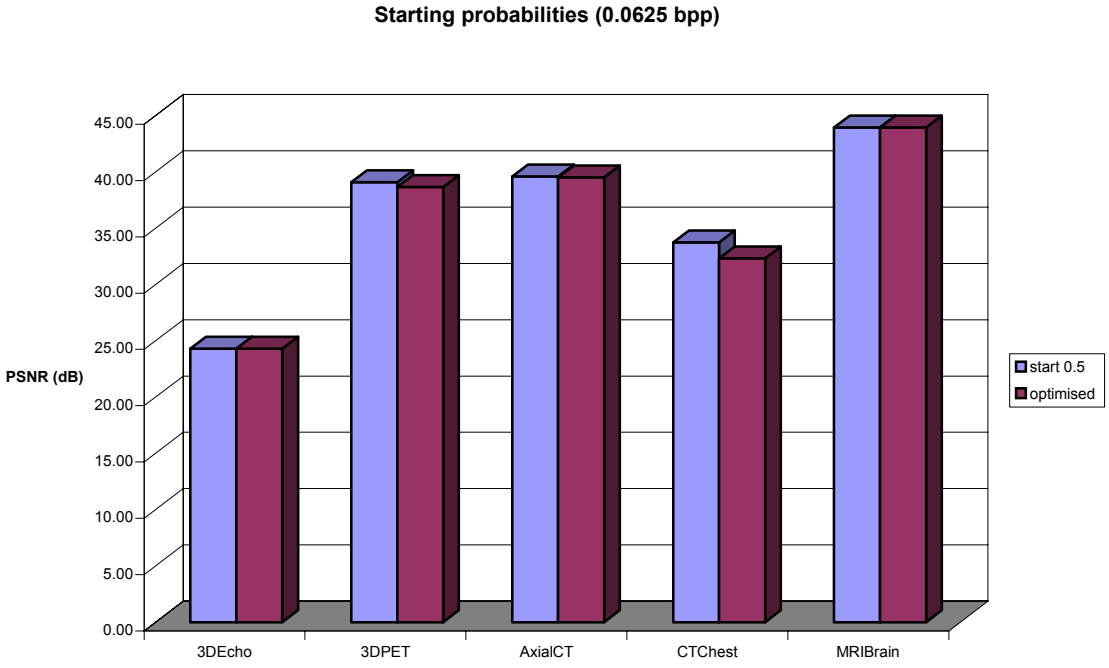


Figure 35: Performance of chosen probabilities against the 0.5 probability case

RESULTS
There is no gain by using these starting probabilities for lossless and lossy coding, in fact there is a slight loss. This is because we have chosen the probabilities of the heaviest pass, causing a sub-optimal performance. We propose a future experiment selecting the probabilities of the first pass that codes a significant amount of pixels.

3.4 Coding with 2D JPEG2000 (VM 7.0)

TARGET
Evaluate the gain obtained with the CS-EBCOT coder compared to the 2D JPEG 2000.

In order to evaluate the gain of the 3D coder to the 2D implementation of the JPEG2000 coder, the test images were spliced into slices with the C program *split.c* to be coded independently. After decompressing the images with the 2D decoder, the resulting slices were merged again with the C program *merge.c*. The final volume was analysed and compared to the one obtained with the 3D coder.

The Verification Model 7.0 the integer wavelet kernel (5,3) as basis for a reversible transforms (identified by *code 1* in the CS-EBCOT coder).

3.4.1 2D JPEG2000 slice by slice

The original test volumes were spliced and transformed to the PGX format in order to be understood by the VM7.0 coder. The VM7.0 coder generated a separate (compressed) file for each slice of the volume.

PGX format

The PGX format just attaches a single text header line of the form “PG <byte order> [+|-]<bit-depth><cols><rows>”, with the binary data appearing immediately after the new line character packed into 1, 2 or 4 bytes per sample, depending upon the value of <bit-depth>. The <byte order> field must be on of the fixed strings “LM” (LSB’s first) or “ML” (MSB’s first) while the optional ‘+’(default) or ‘-‘ character preceding the <bit-depth> field indicates whether the data is signed or not. Any bit-depth from 1 to 32 is acceptable.

The results obtained and compared to the CS-EBCOT coder are the following:

Table 24: 2D JPEG2000 slice by slice against CS-EBCOT coder. Data for lossless coding refers to output bit-rate in bpp, data for lossy coding refers to the PSNR expressed in dB

	3D Echo		**3D PET**		*Axial CT*	
	VM 7.0	**CS-EBCOT**	**VM 7.0**	**CS-EBCOT**	**VM 7.0**	**CS-EBCOT**
lossless	4,287040	3,716886	10,215908	8,611403	4,082576	3,806308
2 bpp	41,75	40,32	51,98	56,53	66,83	**65,57**
1 bpp	35,99	35,81	46,01	49,55	59,51	58,90
0,5 bpp	31,99	31,60	42,48	45,43	54,10	53,93
0,25 bpp	29,48	27,61	40,06	42,67	48,74	49,76
0,125 bpp	27,31	25,90	38,38	40,38	43,65	45,44
0,0625bpp	25,82	22,06	38,38	38,86	38,68	41,14
	CT DICOM		**MRI DICOM**			
	VM 7.0	**CS-EBCOT**	**VM 7.0**	**CS-EBCOT**		
lossless	5,169318	4,980332	4,735544	3,954777		
2	59,48	56,36	62,95	63,66		
1	53,08	50,28	56,42	58,35		
0,5	49,17	46,89	51,70	53,44		
0,25	45,70	43,08	47,81	49,47		
0,125	41,65	39,02	44,56	46,83		
0,0625	38,14	34,50	41,45	43,76		

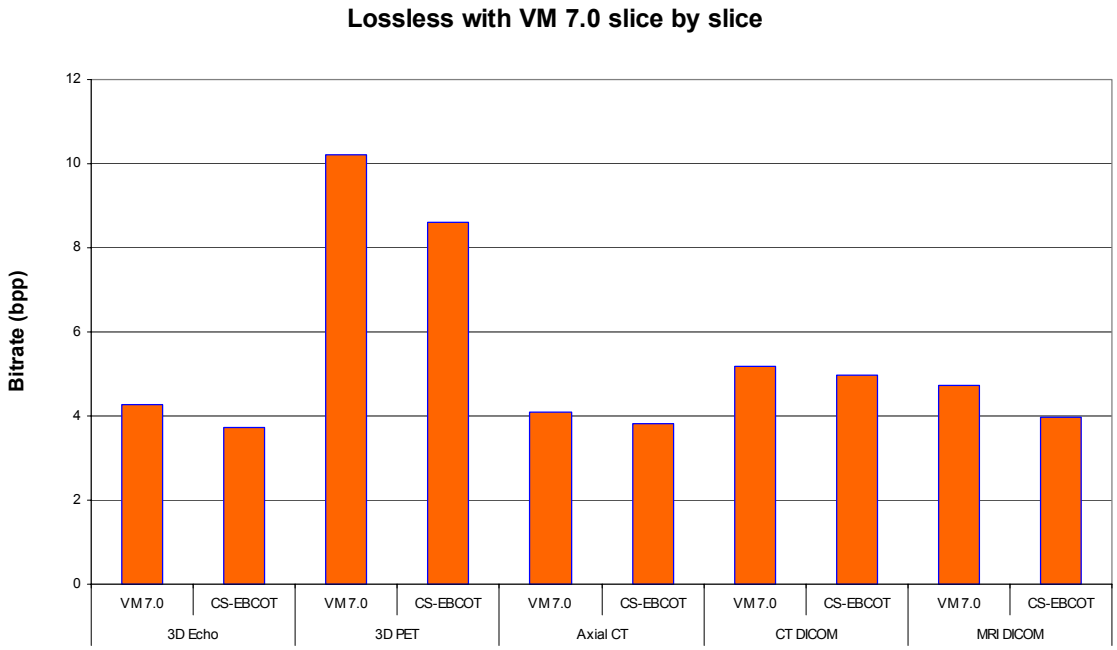


Figure 36: Verification Model 7.0 slice by slice against CS-EBCOT for lossless coding

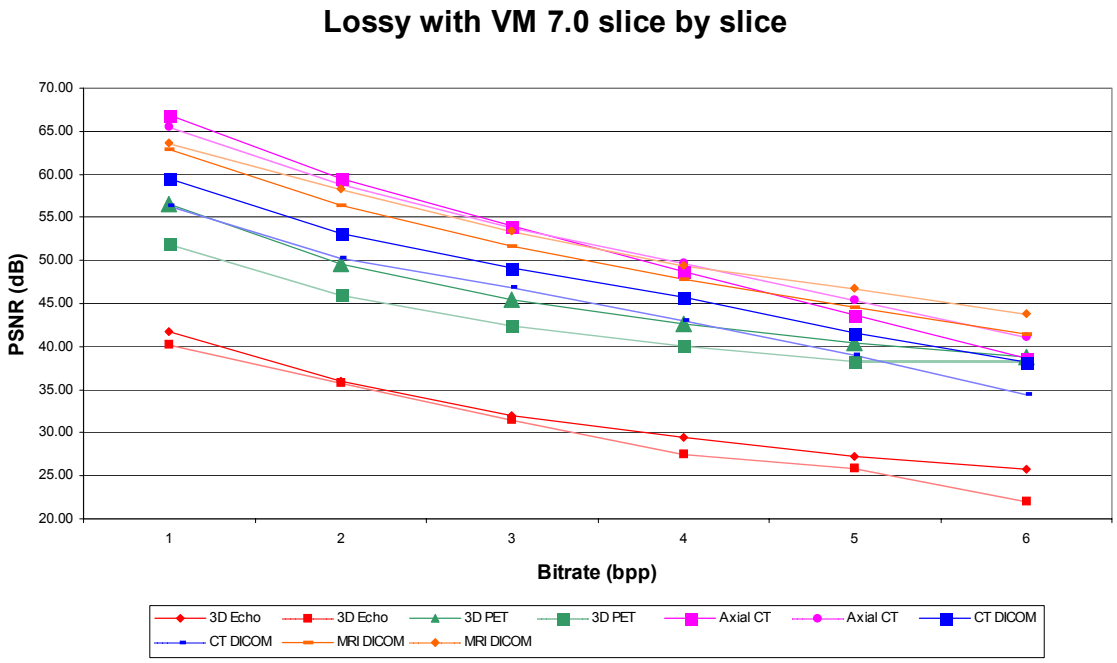


Figure 37: Verification Model 7.0 slice by slice against CS-EBCOT for lossy coding

We were not able to retrieve all data for the 0,0625 bit-rate, since for small images this bit-rate could not be reached by the VM 7.0 (it was so low that even not the header fitted at this bit-rate). At the other hand, VM 7.0 is giving a heavier weight to the wavelet coefficients situated in the lower subbands (related to the normalization policy), a feature which is not included in the present CS-EBCOT implementation.

RESULTS
The 3D version is clearly better than the 2D for the lossless coding. The result is not the same for the lossy tests. This is probably because non-normalized transforms have been used, delivering worse performance at low bit-rates. This is caused by the fact that wrong weights are given at the different subbands, resulting in an overemphasizing of the HF information. The way our coder works even magnifies this problem. At the other hand, a new compressed file has been generated for each slice, so the VM 7.0 files contained a lot of redundancy because of the multiple headers with the same information.

3.4.2 slices as components

The Verification Model 7.0 is able to deal with multi-component images and performing the wavelet transform in the component dimension. This is exactly what the CS-EBCOT coder does. Hence, evaluating this feature could give interesting information. Unfortunately, some bugs were found in the Verification Model 7.0 when trying to deal with multiple components slices were identified as components)Currently, we are trying to locate the problem in collaboration with the editor of the VM software

However, some of the tests could be performed with the following results.

Table 25: Verification Model 7.0 slice as components against CS-EBCOT

	3D Echo		3D PET	
	VM 7.0	**C-EBCOT**	**VM 7.0**	**CS-EBCOT**
lossless	4,278407	3,716886	10,182129	8,611403
2		40,32	52,69	56,53
1		35,81	46,65	49,55
0,5		31,60	43,05	45,43
0,25		27,61	40,65	42,67
0,125		25,90	39,19	40,38
0,0625		22,06	37,88	38,86

Figure 38: Verification Model 7.0 slice as components against CS-EBCOT for lossless coding

Figure 39: Verification Model 7.0 slice as components against CS-EBCOT for lossy coding

RESULTS
In this case, a single compressed file was generated and the results obtained were slightly better than when as many compressed files as slices were generated. This was due that some data (headers, markers), which were repeated in every separate file in the previous experiment a), appeared only once in this unique generated file.

3.5 Scanning pattern

TARGET
Determine which scanning pattern works best.

Two scanning patterns have been implemented in the codec: a traditional 2D JPEG2000 scanning working slice-by-slice and a Morton curve.

The traditional 2D JPEG2000 scanning pattern reads the volume image slice-by-slice and scans each slice by following four sample stripes in the Y direction. After each stripe, the pointer moves to the next one in the increasing direction of X. When a row is finished, a four sample shift is applied in the Y direction. This pattern is repeated till the end of the slice. Next, the procedure is repeated for the succeeding slice.

The Morton Scanning pattern tries to group neighbouring samples as much as possible. This could be an advantage, because by doing this, the big jump from the end of one slice to the beginning of a new one would be avoided, and the arithmetic coder state wouldn't change so quickly.

All tests have been performed with five decomposition levels.

Table 26: Traditional 2D JPEG200 Scanning pattern against 3D Morton Scanning pattern

	filter	lossless (bit-rate)		2 bpp (PSNR)		0.25 bpp (PSNR)	
		2D JPEG	Morton	2D JPEG	Morton	2D JPEG	Morton
3DEcho	3	3.656	3.658	**39.46**	39.45	**26.90**	26.90
3DPET	5	7.966	7.966	**57.60**	53.48	43.45	42.46
AxialCT	5	3.612	3.615	**66.57**	66.54	50.31	50.32
CTChest	3	4.955	4.957	**56.37**	54.52	42.66	41.91
MRIBrain	5	3.718	3.721	**63.86**	63.84	**50.68**	50.67

	filter	0.125 bpp (PSNR)		0.0625 bpp (PSNR)	
		2D JPEG	Morton	2D JPEG	Morton
3DEcho	3	**25.18**	25.11	**24.70**	24.69
3DPET	5	41.34	39.70	39.04	38.55
AxialCT	5	46.22	46.21	39.56	39.52
CTChest	3	38.40	37.55	33.90	32.65
MRIBrain	5	47.19	47.25	43.92	43.91

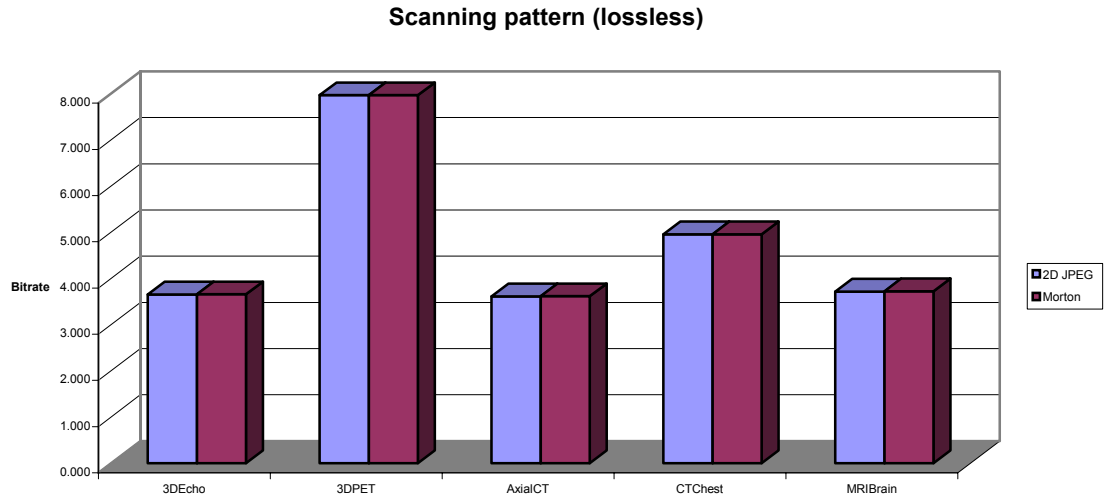


Figure 40: 2D JPEG scanning pattern against 3D Morton scanning pattern for lossless coding

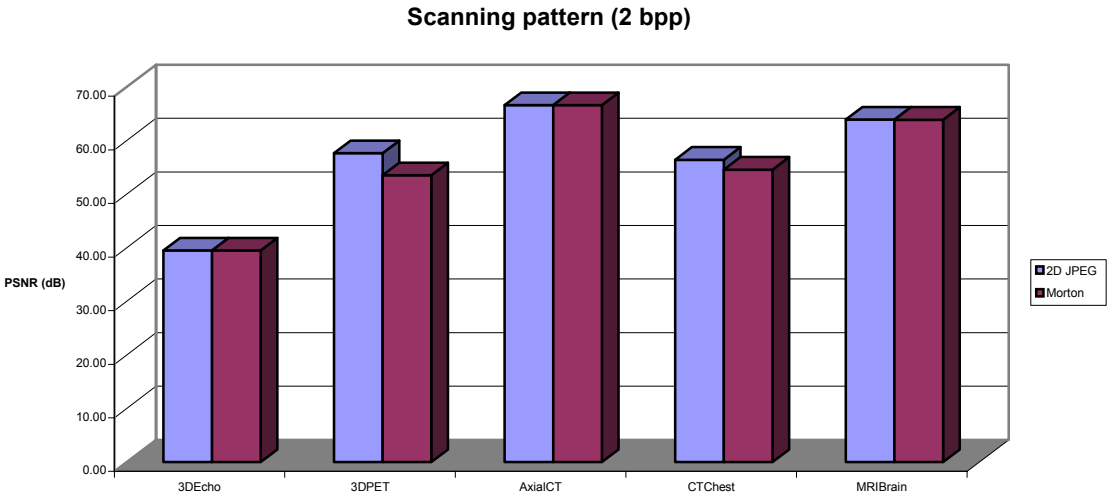


Figure 41: 2D JPEG scanning pattern against 3D Morton scanning pattern for lossy coding

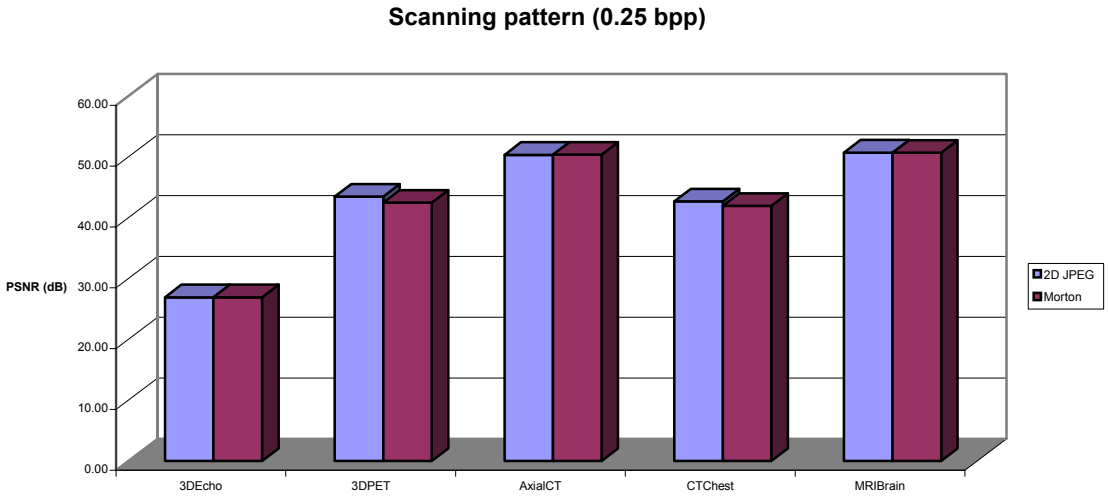


Figure 42: 2D JPEG scanning pattern against 3D Morton scanning pattern for lossy coding

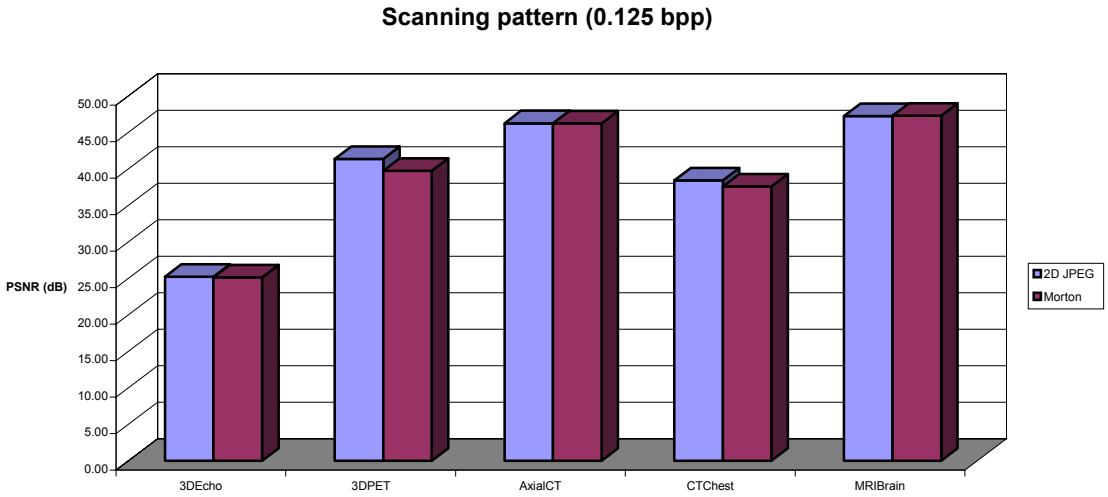


Figure 43: 2D JPEG scanning pattern against 3D Morton scanning pattern for lossy coding

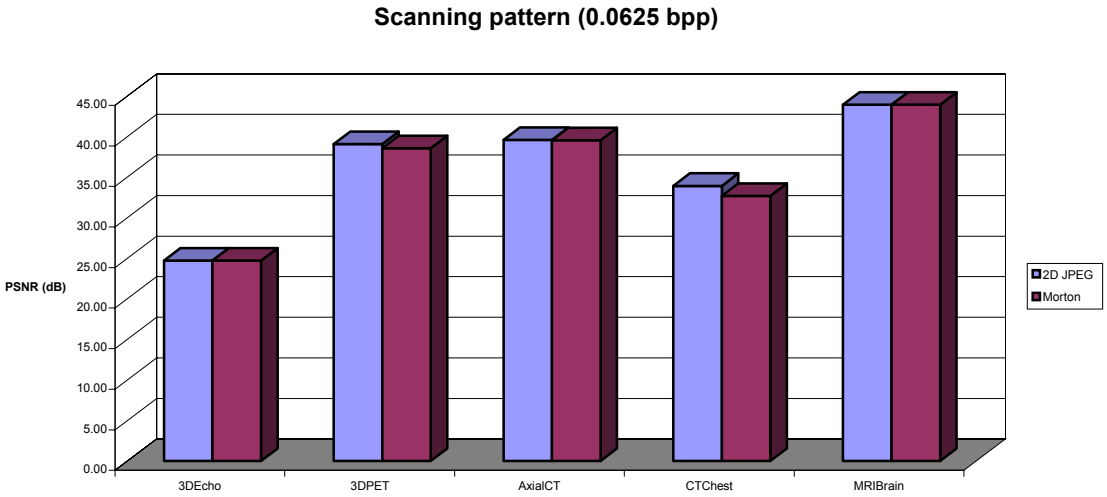


Figure 44: 2D JPEG scanning pattern against 3D Morton scanning pattern for lossy coding

RESULTS
No significance gain has been reached with the Morton scanning.

3.6 Comparative study of CS-EBCOT and other three-dimensional image coders

This experiment was published by Peter Schelkens, Xavier Giro, Joeri Barbarien, Adrian Munteanu, Jan Cornelis and can be found in the bibliography under reference [Sch00b].

It must be stated that in the document the CS-EBCOT coder is referred as 3D JPEG2000 coder.

INTRODUCTION

The increasing use of three-dimensional imaging modalities, like Magnetic Resonance Imaging (MRI), Computer-assisted Tomography (CT), Ultrasound (US), Single Photon Emission Computed Tomography (SPECT) and Positron Emission Tomography (PET), triggers the need for efficient techniques to transport and store the related volumetric data. In a classical approach, the image volume is then considered as composed of multiple slices. These slices are then successively compressed and broadcasted. Contemporary transmission techniques make use of concepts like rate scalability, quality- and resolution scalability. In a 2D world this results in an image being encoded in different quality and/or resolution layers. Also, for volumetric sets the scalability paradigm causes the introduction of multiplexing mechanisms to select from each slice the correct layer(s) to support the actually required QoS layer. However, a disadvantage of the slice-by-slice mechanism is that potential 3D correlations are neglected.

In recent past, 3D DCT based techniques have been proposed, but the weakness of such systems is that they hardly meet the requirements imposed by the scalability paradigm and additionally do not support lossless coding. The latter is extremely important for medical data since it allows to avoid liability problems, and to guarantee the integrity of the medical diagnosis. Hence, we have been looking for other methods that give better support for the above-mentioned requirements. Good candidates are techniques based on wavelet compression. A typical example of 3D wavelet coding is the octave zero-tree based coding [Bil99, Xio99, Kim99, Kim00], which currently tends to deliver the best compression performance. In this report we will present two new approaches for volumetric wavelet coding - Cube-Splitting [Sch00a, Sch00b] and a 3D version of the JPEG 2000 VM - and compare it against an implementation of 3D SPIHT [Kim99, Kim00] and a classical 3D JPEG-based approach for different medical imaging modalities. Furthermore, we evaluate the performance of a selected set of lossless integer lifting kernels.

Cube-Splitting

The proposed coding engine exists out of three main components: a 3D wavelet transform (WT) module, the actual Cube-Splitting module and a context-based arithmetic encoder (CAE). The technique we propose is derived from a 2D square partitioning coder (SQP) [Mun99], which delivers for 2D medical images equivalent lossy coding results as Set Partitioning in Hierarchical Trees (SPIHT) [Sai96b], while it outperforms SPIHT for lossless coding. The coding principle is based on Successive Approximation Quantization (SAQ), with each bit-plane being encoded in the two classical stages: a significance pass and a refinement pass.

During the first significance pass the wavelet coefficients, newly identified as significant, are registered using a recursive tree structure of cubes (

Figure .a-c.). If a cube, having initially the size of the original data volume, contains a significant coefficient, it is spliced in eight sub-cubes. The descendent "significant" cube (or cubes) is then further spliced until the significant coefficients (=pixel nodes) are isolated. The result is an eight-tree structure (

Figure .d). As might be noticed, equal importance weights are given to all the branches. When a significant coefficient is isolated, also its sign - for which two code symbols are preserved - is immediately encoded. At that point, the refinement pass is initiated for the next bit-plane, refining all coefficients marked as significant in the eight-tree. Next, the significance pass is restarted to update the eight-tree by identifying the new significant coefficients for the current bit-plane. During this stage, only the non-significant nodes are re-encoded, and the significant ones are ignored since the decoder already received this information. The complete significance procedure can thus actually be seen as a tree growing process. The described procedure is repeated, until the desired bit-rate is obtained.

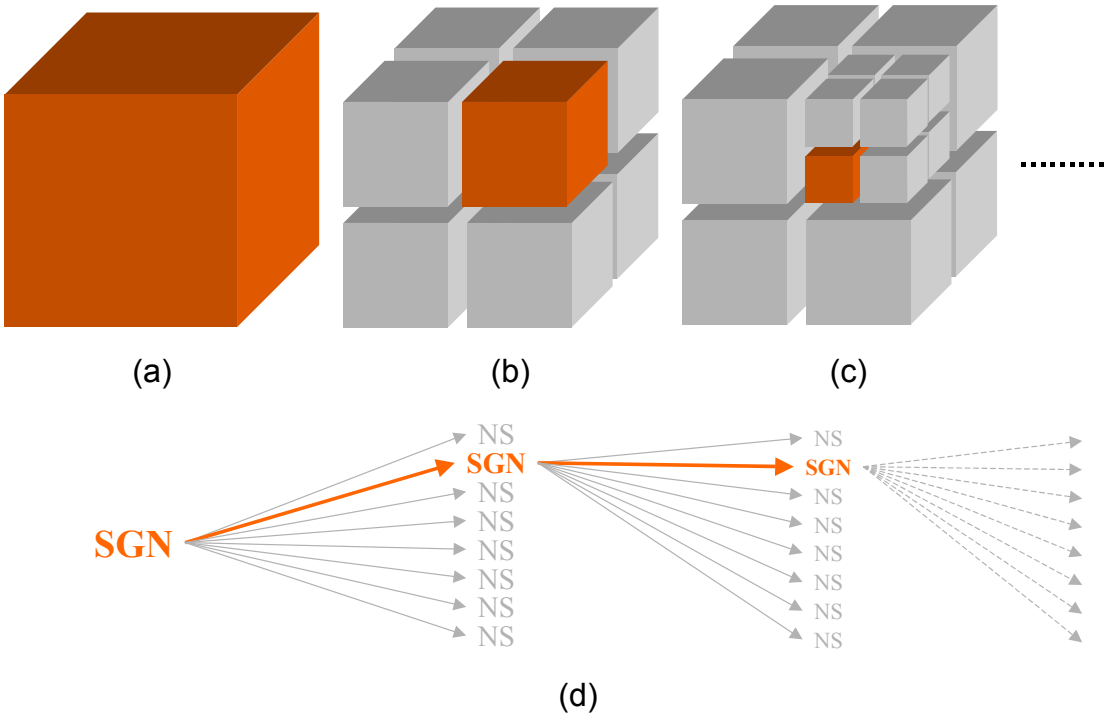


Figure 45 - When a significant wavelet coefficient is encountered, the cube (a) is spliced in eight sub-cubes (b), and further on (c) up to the pixel resolution. The result is an eight-tree structure (d) (SGN = significant node; NS = non-significant node). In the next significance pass, the non-significant nodes are further refined.

To encode the generated coding symbols efficiently, we make use of a simple context-based arithmetic encoder (CAE). The chosen adaptive arithmetic encoder is based on an implementation of the algorithm proposed by I.H. Witten et al. [Wit87]. The context model is extremely simple. For the dominant pass we distinguish four contexts, namely one for the symbols generated for the intermediate cube nodes, one for the pixel nodes having non-significant neighbors for the previous threshold in the SAQ process, one for the pixel nodes having at least one significant neighbor for the previous threshold and finally one for encoding the sign of the isolated significant pixel nodes. Only two contexts are used for the refinement pass: one for the pixel nodes having non-significant neighbors for the previous threshold, one for the pixel nodes having at least one significant neighbor for the previous threshold.

3D SPIHT

The 3D SPIHT encoder [Kim00] (an early version of the latter has already proven to beat the performance of a context-based zero-tree coder [Xio99]) was equipped with the same wavelet transform front-end as the Cube-Splitting coder, as well as the same arithmetic encoding back-end [Wit87]. The SPIHT implementation in this study uses balanced 3D orientation trees, i.e. the same amount of recursive wavelet decompositions is required for all spatial orientations (x, y and z). If this is not respected, several tree nodes will not refer to or be linked with the same spatial location, consequently destroying the correlation between different tree-nodes and reducing the compression performance. Solutions have been proposed utilizing unbalanced spatio-temporal orientation trees in the context of video coding [Kim98], though the introduction of it increases the implementation complexity and/or reduces the flexibility of the coding engine. Additionally, context-based arithmetic coding was applied for the significance pass using a lot of different context-models for the tree nodes (each tree node representing 2x2x2 pixels). The context identification is based on the significance behavior of the eight node pixels and their descendents (as in [Xio99]).

3D JPEG

The 3D JPEG-based coder is composed of a discrete cosine transform (DCT) followed by a scalar quantization stage and finally a combination of run-length coding and adaptive arithmetic encoding. The principle is simple: the volume is divided in cubes of 8x8x8 pixels and each cube is separately DCT-transformed; as it is the case for a classical JPEG-coder. The DCT-coefficients are then uniformly quantized as a consequence of the application of a scaling factor, called the quality factor. Next, the quantized DCT-coefficients are scanned using a 3D space-filling curve, i.e. a 3D instantiation of the 2D Morton-curve [Mor66], to allow for the grouping of zero-valued coefficient and hence to improve the performance of the run-length coding. We opted for this curve due to its simplicity compared to that of 3D zig-zag curves [Lee97]. The non-zero coefficients are encoded using the same classification system as for JPEG. The coefficient values are grouped in 16 main magnitude classes (ranges), which are subsequently encoded with an arithmetic encoder [Wit87]. Finally, the remaining bits to refine the coefficients within one range are added without further entropy coding.

Experiments

In the experiments the coders are evaluated making use of five medical data sets: one reconstructed ultrasound sequence, one PET volume, two CT scans and one MRI scan (see *Table 27*).

Table 27 – Medical test data sets

Name	**Size (WxHxD)**	**Bit-range (bpp)**	**Resolution (wxhxd in mm)**	**Imaging Device**	**Details**
US	256x256x256	8	1 x 1 x 1	Ultrasound	Prostate
PET	128x128x128	15	1 x 1 x 1	PET	Brain
CT 1 [#]	512x512x100	12	1 x 1 x 1	CT	Axial scan of female cadaver brain (slice 100-199 of the HVP set)
CT 2 [##]	512x512x44	12	0.66 x 0.66 x 5-10	CT	Helical scan of normal chest and mediastinum
MRI [##]	256x256x200	12	0.86 x 0.86 x 0.80	MRI	T1 weighted field echo 3D volume scan of normal brain

[#] Visible Human Project data set (http://www.nlm.nih.gov/research/visible/), [##] DICOM test set

Lossless coding performance was evaluated for the Cube-Splitting, the 3D SPIHT and the 3D JPEG2000 methodologies. We did not include the 3D JPEG coder in the test due

to the lossy character of its DCT front-end. For all the tests performed for lossless coding (as well as for lossy coding), we applied a 4-level wavelet transform (instead of a 5-level one) in all spatial directions on the PET and the CT 2 images, since both images have a limited size in the direction of the z-axis. Hence, we preferred to reduce in that case also the amount of the decompositions in the x- and y-directions to avoid the destruction of the correlations within the spatial orientation trees of the 3D SPIHT coder, due to the unbalanced character of the latter. It is evident that the Cube-Splitting and the 3D JPEG2000 coders are not limited by such drawbacks, but to ensure a fair comparison, we applied the same restriction for them too. The other images, US, CT1 and MRI, were processed with a 5-level wavelet transform.
Table 28 displays the lossless coding results. Out of this table we generated two extra tables to enable an easier evaluation. *Table 29* shows for each test volume and for each coding technique, the increase of the bit-rate in terms of percentage for the utilized wavelet kernel compared to the optimal kernel. Although it is clear that the results are data and filter kernel dependent, it is possible to draw some general conclusions. In order of performance, we can state that the 13x11, 9x7, 5x11, 13x7 and S+P give the best results, closely followed by a second group, i.e. the 5x3 and 9x3 kernels. Only the S-kernel was performing poorly. From *Table 30* showing for each test volume and for each wavelet kernel, the increase of the bit-rate in terms of percentage for the applied coding technique in comparison to the optimal coder, we can conclude that the 3D JPEG2000 is always outperforming the Cube-Splitting coder and the 3D SPIHT coding engine. While the Cube-Splitting technique is usually compared to 3D SPIHT, although we have to note too that the differences are rather small.

For *lossy coding* the story becomes more complicated. In this evaluation also the 3D JPEG coder is included, and similar tables are generated (see *Table 31-48*). However, the latter are now made up for PSNR measurements at six different bit-rates: 2, 1, 0.5, 0.25, 0.125 and 0.0625 bits-per-pixel (bpp). At low bit-rates we notice that especially the 9x3, 5x11, 9x7 and 5x3 kernels are performing well, closely followed by the 13x7 and 13x11 kernels. At higher bit-rates the latter are moving towards the other ones, which is logic since they were among the best filters for lossless coding. Notice also that the 5x11 kernel seems to be the most stable one, delivering an excellent performance over the complete lossy-to-lossless range. These results coincide largely with the outcome of [Bil99, Ada00]. Concerning the performance of the three coders we observe at high bit-rates (1-2 bpp) the best performance for the 3D JPEG encoder, followed by the 3D JPEG2000, 3D SPIHT and the Cube-Splitting coder. At intermediate bit-rates (0.5 – 0.25 bpp) the performance of the 3D JPEG coder is decreasing fast, and 3D SPIHT, together with 3D JPEG2000, are taking over the lead. And finally, figures for the lowest bit-rates (0.125 – 0.0625 bpp) illustrate a tendency slightly in the advantage of the 3D JPEG2000 and Cube-Splitting coders. Figure and

Figure illustrate the visual performance of the different techniques, while using a 4-level decomposition with a 9x3 kernel, at three different bit-rates. Although, it is difficult to distinguish the difference in performance between the Cube-Splitting and the 3D SPIHT wavelet coders, the 3D JPEG2000 images depict less blurring and sharper edges at lower bit-rates. The poorer performance of the 3D JPEG coder is clear at high compression rates.

Acknowledgements

This research was funded by the Institute for the Promotion of Innovation by Science and Technology in Flanders, Belgium (IWT) within the Information Technology Action II Programme (project: “Image Processing for Integrated and Immersive Visualization”; Project number: IWT-980302).
The authors would also like to thank William Pearlman for his support in relation to the 3D SPIHT coder.

References

[Ada00] M.D. Adams, F. Kossentini, “Reversible integer-to-integer wavelet transforms for image compression: performance evaluation and analysis”, *IEEE Trans. on Image Processing*, Vol. 9, No.6, pp.1010-1024, June 2000.
[Ade87] E.H. Adelson, E. Simoncelli, R. Hingorani, “Orthogonal pyramid transforms for image coding”, *Proc. SPIE Visual Communications and Image Processing II*, Vol. 845, pp.50-58, 1987.

[Bil00] A. Bilgin, G. Zweig, M.W. Marcelin, “Three-dimensional compression with integer wavelet transforms”, *Applied Optics*, Vol. 39, No. 11, pp.1799-1814, April 2000.

[Cal98] A.R. Calderbank, I. Daubechies, W. Sweldens, B.-L. Yeo, “Wavelet transforms that map integers to integers”, *J. Appl. Computa. Harmonics Anal.*, Vol. 5, pp.332-369, 1998.

[Dew97] S. Dewitte, J. Cornelis, “Lossless integer wavelet transform”, *IEEE Signal Processing Letters*, Vol. 4, No.6, pp.158-160, June 1997.

[Jpg00] “JPEG2000 Verification Model VM7 – Technical Description”, ISO/IEC JTC1/SC29/WG1, WG1N1684, April 2000.

[Kim00] Y.S. Kim, W.A. Pearlman, "Stripe-Based SPIHT Lossy Compression of Volumetric Medical Images for Low Memory Usage and Uniform Reconstruction Quality," *Proc. of IEEE Int. Conf. on Acoustics, Speech and Signal Processing (ICASSP 2000)*, Istanbul, Turkey, June 2000.

[Kim98] B.-J. Kim, W.A. Pearlman, “Low-delay embedded 3-D wavelet color video coding with SPIHT”, *Proc. SPIE*, Vol. 3309, pp. 955-964, 1998.

[Kim99] Y. Kim, W.A. Pearlman, “Lossless volumetric medical image compression”, *Proc. SPIE Conference on Applications of Digital Image Processing XXII*, Vol.3808, pp. 305-312, July 1999.

[Lee98] M.C. Lee, R.K.W. Chan, D.A. Adjeroh, “Quantization of 3D-DCT coefficients and scan order for video compression”, *Journal of Visual Communications and Image Representation*, Vol. 8, No.4, pp.405-422, 1997.

[Mor66] G.M. Morton, "A computer oriented geodetic data base and a new technique in file sequencing", IBM Ltd., Ottawa, Canada, 1966.

[Mun99] A. Munteanu, J. Cornelis, G. Van der Auwera, P. Cristea, “Wavelet-based lossless compression scheme with progressive transmission capability”, *International Journal of Imaging Systems and Technology*, Vol. 10, No. 1, pp. 76-85, January 1999.

[Sai96a] A. Said, W. Pearlman, “An image multiresolution representation for lossless and lossy compression”, *IEEE Trans. on Image Processing*, Vol. 5, pp.1303-1310, 1996.

[Sai96b] A. Said and W. Pearlman, “ A new fast and efficient image codec based on set partitioning in hierarchical trees”, *IEEE Trans. on Circuits and Systems*, Vol.6, pp. 243-250, March 1996.

[Sch00a] P. Schelkens, J.Barbarien, J. Cornelis, “Volumetric data compression based on cube-splitting”, *Proc. of 21st Symposium on Information Technology in the Benelux*, Vol.21, pp.93-100, May 2000.

[Sch00b] P. Schelkens, J. Barbarien, J. Cornelis, “Compression of Volumetric Medical Data based on cube-splitting”, *Proc. SPIE Conference on Applications of Digital Image Processing XXIII*, Vol.4115, July 2000.

[Wit87] I.H. Witten, R.M. Neal, J.G. Cleary, “Arithmetic coding for data compression”, *Communications of the ACM*, Vol. 30, No.6, pp. 520-540, June 1987.

[Xio98] Z. Xiong, K. Ramchandran, M.T. Orchard, “Wavelet packet image coding using space-frequency quantization”, *IEEE Trans. on Image Processing*, Vol. 7, pp. 892-898, June 1998.

[Xio99] Z. Xiong, X. Wu, D.Y. Yun, W.A. Pearlman, “Progressive coding of medical volumetric data using three-dimensional integer wavelet packet transform”, *Proc. SPIE Conference on Visual Communications*, Vol. 3653, pp. 327-335, January 1999.

Table 28 – Lossless compression results for a 4-level (PET, CT2) and a 5-level (US, CT1, MRI) wavelet decomposition.

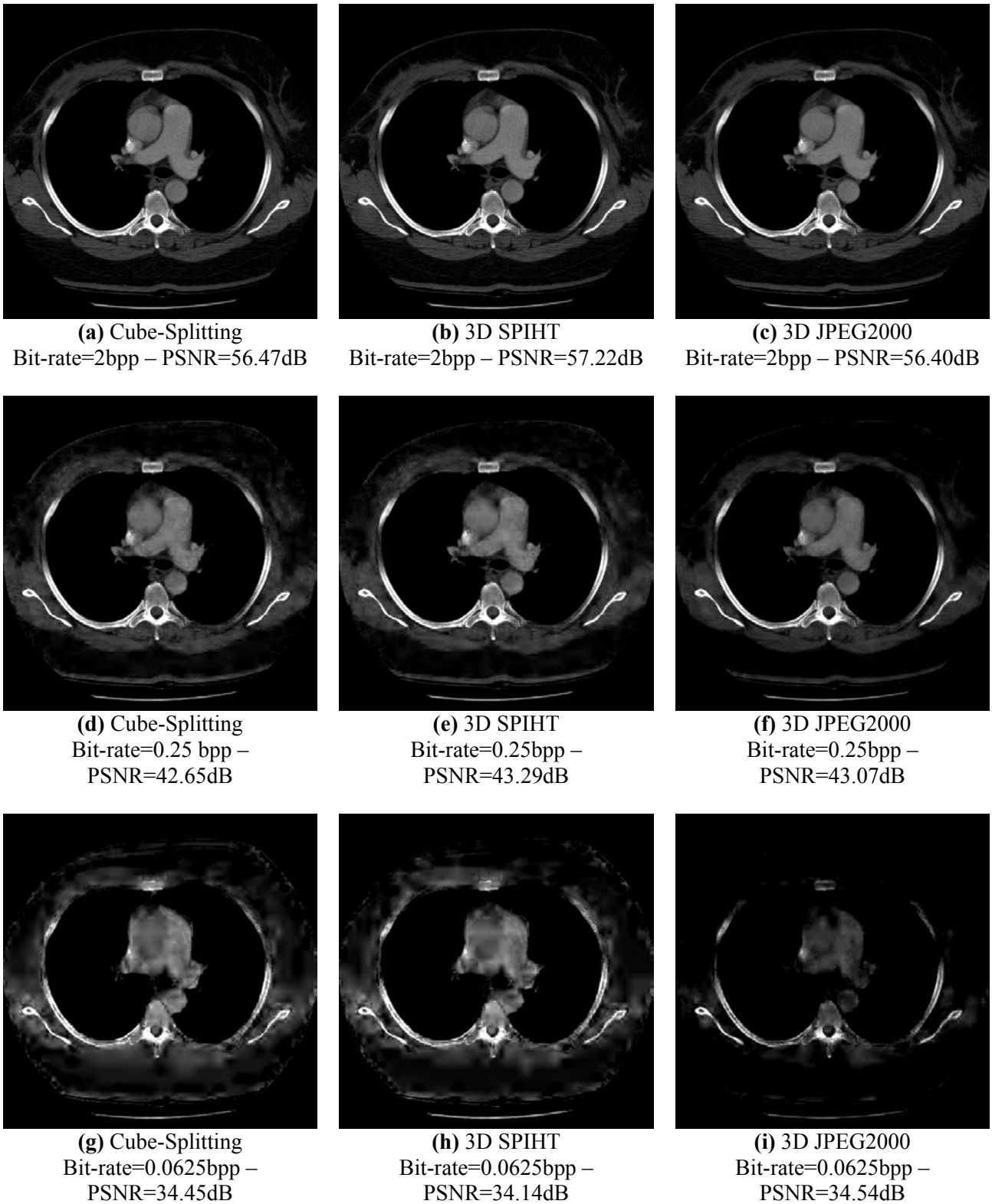
	Lossless Coding Results (in bpp)														
	3D Cube -Splitting					3D SPIHT					3D JPEG2000				
	US	PET	CT 1	CT 2	MRI	US	PET	CT 1	CT 2	MRI	US	PET	CT 1	CT 2	MRI
5x3	3.7649	8.7558	3.8612	5.0331	4.0296	3.7735	9.1558	3.9286	5.1951	4.0529	3.7169	8.6112	3.8063	4.9805	3.9548
S	4.1916	9.7236	4.4230	5.4135	4.5652	4.1789	9.8688	4.3971	5.3699	4.5358	4.0953	9.5620	4.3272	5.3093	4.4538
9x7	**3.7096**	8.3188	**3.7137**	**5.0169**	3.8757	**3.7182**	8.9805	3.8087	5.2796	3.9080	**3.6560**	8.1576	3.6478	**4.9549**	3.7757
9x3	3.7886	8.7929	3.8955	5.0648	4.0587	3.8031	9.3013	3.9786	5.2716	4.0885	3.7406	8.6487	3.8417	5.0117	3.9827
13x11	**3.7097**	8.9191	3.7889	5.3491	**3.8622**	3.7204	**8.1429**	**3.6813**	5.3491	**3.8265**	3.6571	**7.9653**	**3.6119**	4.9708	**3.7180**
5x11	3.7163	**8.2948**	3.7226	5.0170	3.8819	3.7296	8.9656	3.8285	5.3065	3.9181	3.6671	8.1310	3.6589	4.9569	3.7903
2x6	3.8354	9.2927	4.0455	5.3063	4.1605	3.8429	8.8791	3.9829	5.3063	4.1412	3.7691	8.7031	3.8897	5.0642	4.0210
S+P	3.7283	8.9292	3.8343	5.1845	3.9238	3.7376	8.5193	3.7663	**5.1845**	3.8955	3.6835	8.3526	3.7102	4.9619	3.8275
13x7	3.7138	8.3391	3.7337	5.0323	3.8989	3.7253	9.0940	3.8420	5.3468	3.9380	3.6593	8.1788	3.6649	4.9688	3.7953

Table 29 – Best performing transform per coding technique and per test image for lossless coding for a 4-level (PET, CT2) and a 5-level (US, CT1, MRI) wavelet decomposition. The figures depict the deviation in percent from the optimal transform.

	Best Transform per Coding Technique (in %)															
	3D Cube -Splitting					3D SPIHT					3D JPEG2000					
	US	PET	CT 1	CT 2	MRI	US	PET	CT 1	CT 2	MRI	US	PET	CT 1	CT 2	MRI	AV
5x3	1.49	5.56	3.97	0.32	4.33	1.49	12.44	6.72	0.20	5.92	1.67	8.11	5.38	0.52	6.37	4.30
S	12.99	17.23	19.10	7.90	18.20	12.39	21.19	19.44	3.58	18.54	12.02	20.05	19.80	7.15	19.79	15.29
9x7	**0.00**	0.29	**0.00**	**0.00**	0.35	**0.00**	10.29	3.46	1.83	2.13	**0.00**	2.41	1.00	**0.00**	1.55	1.55
9x3	2.13	6.01	4.89	0.95	5.09	2.28	14.22	8.08	1.68	6.85	2.31	8.58	6.36	1.15	7.12	5.18
13x11	**0.00**	7.53	2.02	6.62	**0.00**	0.06	**0.00**	**0.00**	3.18	**0.00**	0.03	**0.00**	**0.00**	0.32	**0.00**	1.32
5x11	0.18	**0.00**	0.24	0.00	0.51	0.30	10.10	4.00	2.35	2.40	0.30	2.08	1.30	0.04	1.94	1.72
2x6	3.39	12.03	8.93	5.77	7.72	3.35	9.04	8.19	2.35	8.23	3.09	9.26	7.69	2.21	8.15	6.63
S+P	0.50	7.65	3.25	3.34	1.59	0.52	4.62	2.31	**0.00**	1.80	0.75	4.86	2.72	0.14	2.94	2.47
13x7	0.11	0.53	0.54	0.31	0.95	0.19	11.68	4.37	3.13	2.91	0.09	2.68	1.47	0.28	2.08	2.09

Table 30 - Best performing coding technique per transform and per test image for lossless coding for a 4-level (PET, CT2) and a 5-level (US, CT1, MRI) wavelet decomposition. The figures depict the deviation in percent from the optimal coding technique.

	Best Coding Technique per Transform (in %)														
	3D Cube -Splitting					3D SPIHT					3D JPEG2000				
	US	PET	CT 1	CT 2	MRI	US	PET	CT 1	CT 2	MRI	US	PET	CT 1	CT 2	MRI
5x3	1.29	1.68	1.44	1.06	1.89	1.52	6.32	3.21	4.31	2.48	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
S	2.35	1.69	2.22	1.96	2.50	2.04	3.21	1.62	1.14	1.84	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
9x7	1.47	1.98	1.81	1.25	2.65	1.70	10.09	4.41	6.55	3.50	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
9x3	1.28	1.67	1.40	1.06	1.91	1.67	7.55	3.56	5.19	2.66	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
13x11	1.44	11.97	4.90	7.61	3.88	1.73	2.23	1.92	7.61	2.92	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
5x11	1.34	2.01	1.74	1.21	2.42	1.70	10.26	4.63	7.05	3.37	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
2x6	1.76	6.77	4.01	4.78	3.47	1.96	2.02	2.40	4.78	2.99	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
S+P	1.22	6.90	3.35	4.49	2.52	1.47	2.00	1.51	4.49	1.78	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
13x7	1.49	1.96	1.88	1.28	2.73	1.81	11.19	4.83	7.61	3.76	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
AV	1.52	4.07	2.53	2.74	2.66	1.73	6.10	3.12	5.41	2.81	0.00	0.00	0.00	0.00	0.00



(a) Cube-Splitting Bit-rate=2bpp – PSNR=56.47dB

(b) 3D SPIHT Bit-rate=2bpp – PSNR=57.22dB

(c) 3D JPEG2000 Bit-rate=2bpp – PSNR=56.40dB

(d) Cube-Splitting Bit-rate=0.25 bpp – PSNR=42.65dB

(e) 3D SPIHT Bit-rate=0.25bpp – PSNR=43.29dB

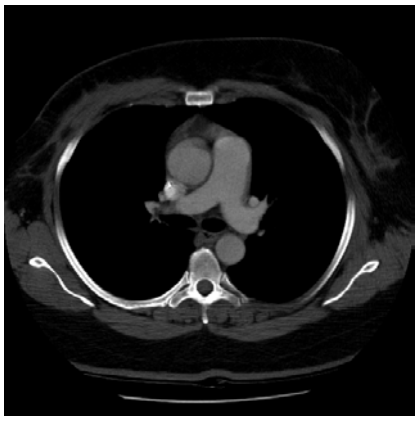
(f) 3D JPEG2000 Bit-rate=0.25bpp – PSNR=43.07dB

(g) Cube-Splitting Bit-rate=0.0625bpp – PSNR=34.45dB

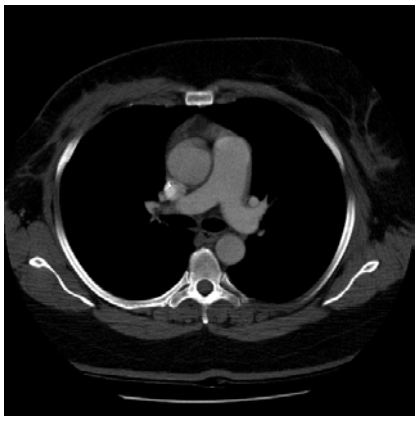
(h) 3D SPIHT Bit-rate=0.0625bpp – PSNR=34.14dB

(i) 3D JPEG2000 Bit-rate=0.0625bpp – PSNR=34.54dB

Figure 46 – Visual compression results for CT 2. The results were obtained with a 9x3 filter kernel and 4 levels of wavelet decomposition. The greyscale values in the interval [910,1604] are visualized.



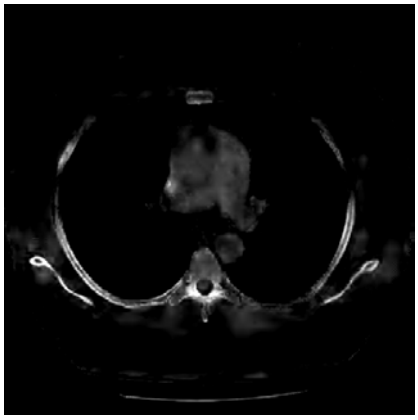
(c) 3D JPEG2000
Bit-rate=2bpp
PSNR=56.40dB



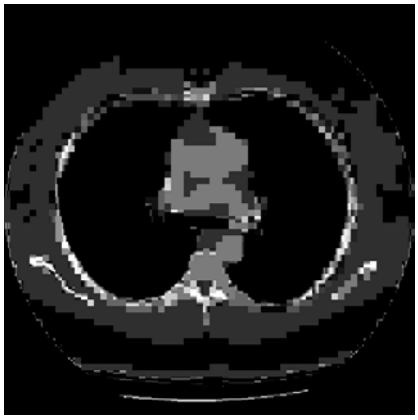
(d) 3D JPEG
Bit-rate=2bpp
PSNR=56.74dB

(f) 3D JPEG2000
Bit-rate=0.25bpp
PSNR=43.07dB

(e) 3D JPEG
Bit-rate=0.25bpp
PSNR=40.55dB



(i) 3D JPEG2000
Bit-rate=0.0625bpp
PSNR=34.54dB



(f) 3D JPEG
Bit-rate=0.0625bpp
PSNR=30.85dB

Figure 47 – Visual compression results for CT 2. The results for 3D JPEG2000 were obtained with a 9x3 filter kernel and 4 levels of wavelet decomposition. The greyscale values in the interval [910,1604] are visualized.

Table 31 - Lossy compression results (2 bpp) for a 4-level (PET, CT2) and a 5-level (US, CT1, MRI) wavelet decomposition

	Lossy Coding Results for 2bpp (in dB)														
	3D Cube -Splitting					3D SPIHT					3D JPEG2000				
	US	PET	CT 1	CT 2	MRI	US	PET	CT 1	CT 2	MRI	US	PET	CT 1	CT 2	MRI
5x3	39.71	56.91	63.95	56.40	63.10	40.78	57.96	64.80	57.38	64.26	40.32	56.53	65.57	56.36	63.66
S	35.83	48.96	57.41	50.64	56.78	36.38	48.38	59.53	51.87	58.18	35.43	50.00	60.98	52.99	56.81
9x7	38.99	57.65	62.60	55.88	62.63	40.40	58.45	64.06	56.47	64.15	39.46	57.39	66.35	56.37	63.97
9x3	39.84	56.94	63.94	56.47	63.39	40.87	57.97	64.33	57.22	64.31	40.63	56.52	65.62	56.40	63.88
13x11	38.54	57.26	62.47	55.44	62.29	39.83	58.29	63.63	56.01	63.73	38.96	57.60	66.57	56.38	63.86
5x11	39.90	58.36	63.01	56.40	63.18	40.99	59.36	64.55	57.09	64.68	40.17	57.97	66.43	56.51	64.48
2x6	35.47	54.30	60.72	53.30	60.71	36.86	54.17	60.78	53.94	61.06	37.34	53.64	64.26	54.96	59.65
S+P	35.41	54.49	60.28	53.93	60.55	36.80	54.24	61.43	53.93	61.53	37.15	55.91	65.34	54.59	62.26
13x7	39.18	57.76	62.22	56.09	62.93	40.29	58.34	64.06	56.63	64.09	39.58	57.43	66.43	56.41	64.13
3D JPEG	43.26	58.46	65.49	56.74	65.51										

Table 32 - Lossy compression results (1 bpp) for a 4-level (PET, CT2) and a 5-level (US, CT1, MRI) wavelet decomposition

	Lossy Coding Results for 1bpp (in dB)														
	3D Cube -Splitting					3D SPIHT					3D JPEG2000				
	US	PET	CT 1	CT 2	MRI	US	PET	CT 1	CT 2	MRI	US	PET	CT 1	CT 2	MRI
5x3	34.77	49.65	57.96	50.93	57.49	36.80	50.31	58.49	51.55	58.21	35.81	49.55	58.90	50.28	58.35
S	29.84	43.23	50.88	45.12	50.82	29.92	42.74	52.56	46.17	51.61	30.58	45.31	53.86	46.48	50.41
9x7	35.55	49.74	57.11	50.31	57.02	35.99	49.79	58.44	50.79	57.80	34.95	50.44	60.03	50.05	58.51
9x3	34.91	49.74	58.20	51.04	57.64	36.74	50.28	58.58	51.63	58.42	35.87	49.81	59.11	50.24	58.13
13x11	35.22	49.66	56.80	49.92	56.74	35.47	49.48	58.20	50.36	57.80	34.46	50.39	60.27	49.92	58.55
5x11	35.91	50.37	58.14	50.83	57.90	36.62	50.71	59.07	51.33	58.54	35.48	50.34	60.04	50.25	58.82
2x6	30.49	47.56	55.42	48.48	55.14	32.41	46.79	55.47	48.65	55.38	32.05	46.65	56.95	48.20	53.61
S+P	30.11	47.60	55.11	48.59	55.03	32.57	46.79	55.82	48.45	55.30	31.80	47.24	58.91	48.23	54.02
13x7	35.73	49.93	57.68	50.49	57.49	35.97	49.95	58.69	50.96	58.02	34.72	50.43	60.18	49.97	58.09
3D JPEG	37.02	50.24	57.94	50.26	58.49										

Table 33 - Lossy compression results (0.5 bpp) for a 4-level (PET, CT2) and a 5-level (US, CT1, MRI) wavelet decomposition

	Lossy Coding Results for 0.5 bpp (in dB)														
	3D Cube -Splitting					3D SPIHT					3D JPEG2000				
	US	PET	CT 1	CT 2	MRI	US	PET	CT 1	CT 2	MRI	US	PET	CT 1	CT 2	MRI
5x3	32.33	44.84	52.78	46.75	52.77	32.53	45.11	53.62	47.32	53.32	31.60	45.43	53.93	46.89	53.44
S	25.50	41.29	46.05	40.93	45.67	27.54	40.93	47.08	43.65	48.00	27.22	41.33	48.41	42.23	44.28
9x7	31.39	44.39	52.26	45.95	52.28	31.21	44.36	53.48	46.39	53.31	30.89	46.03	54.79	46.38	54.27
9x3	32.64	45.06	53.12	46.86	53.06	32.82	45.24	53.63	47.37	53.74	31.62	45.56	54.16	46.75	53.66
13x11	30.99	43.99	52.49	45.52	51.75	30.74	43.94	53.05	45.89	52.83	30.60	46.24	54.91	46.27	54.50
5x11	32.29	44.99	52.98	46.53	52.95	32.41	45.14	54.16	47.01	54.18	31.28	46.10	54.59	46.59	53.80
2x6	29.43	42.64	50.12	44.46	50.08	29.04	41.91	50.27	44.39	49.87	28.31	43.93	51.92	44.37	48.90
S+P	29.09	42.32	49.83	44.02	49.56	28.69	41.90	51.07	43.95	50.11	28.22	44.94	53.70	44.09	51.47
13x7	31.78	44.70	52.41	46.07	52.32	31.59	44.62	53.50	46.50	53.35	30.52	46.38	54.82	46.44	53.56
3D JPEG	32.58	44.62	52.05	45.82	52.73										

Table 34 - Lossy compression results (0.25 bpp) for a 4-level (PET, CT2) and a 5-level (US, CT1, MRI) wavelet decomposition

	Lossy Coding Results for 0.25 bpp (in dB)														
	3D Cube -Splitting					3D SPIHT					3D JPEG2000				
	US	PET	CT 1	CT 2	MRI	US	PET	CT 1	CT 2	MRI	US	PET	CT 1	CT 2	MRI
5x3	28.06	42.53	48.11	42.51	48.34	28.26	42.95	48.41	43.10	49.11	27.61	42.67	49.76	43.08	49.47
S	24.71	37.30	40.27	38.89	44.36	24.43	37.46	40.29	39.49	43.79	24.42	39.37	39.73	37.69	39.95
9x7	26.65	42.55	47.77	41.55	50.15	28.54	42.74	48.70	41.93	50.44	26.90	42.94	50.51	42.66	49.41
9x3	28.38	41.04	48.23	42.65	48.51	28.47	43.00	48.50	43.29	49.58	27.56	42.77	49.21	43.07	49.77
13x11	26.20	42.45	48.04	41.05	50.10	28.09	42.44	48.88	41.43	50.29	26.56	43.45	50.31	42.40	50.68
5x11	27.75	42.82	48.32	42.31	48.71	28.03	43.08	49.49	42.69	51.03	27.16	43.03	50.49	42.95	49.97
2x6	25.03	41.15	45.12	39.95	45.06	24.89	40.28	44.73	39.59	47.45	26.35	42.11	46.86	40.15	43.54
S+P	24.63	41.41	45.54	39.50	48.66	26.35	40.66	45.70	39.25	48.28	25.95	42.54	47.10	40.08	46.72
13x7	27.07	42.78	47.42	41.68	49.51	27.91	42.85	48.48	42.14	50.54	26.65	43.24	50.33	42.61	49.53
3D JPEG	28.58	41.10	45.68	40.55	48.18										

Table 35 - Lossy compression results (0.125 bpp) for a 4-level (PET, CT2) and a 5-level (US, CT1, MRI) wavelet decomposition

	Lossy Coding Results for 0.125 bpp (in dB)														
	3D Cube -Splitting					3D SPIHT					3D JPEG2000				
	US	PET	CT 1	CT 2	MRI	US	PET	CT 1	CT 2	MRI	US	PET	CT 1	CT 2	MRI
5x3	27.38	39.83	43.17	37.96	46.85	27.02	39.88	43.12	38.59	47.00	25.90	40.38	45.44	39.02	46.83
S	24.28	36.74	34.10	34.83	40.02	23.89	36.95	34.99	34.91	37.27	23.62	36.15	32.86	33.31	39.02
9x7	26.25	39.17	42.59	37.61	46.38	25.85	38.96	43.29	37.84	46.00	25.18	41.25	46.16	38.40	47.28
9x3	27.63	40.10	43.26	38.19	46.89	27.34	40.11	43.27	38.67	47.16	25.82	40.81	45.44	39.09	45.85
13x11	25.85	38.83	42.55	37.14	45.94	25.40	38.52	43.60	37.34	45.59	24.69	41.34	46.22	38.14	47.19
5x11	27.18	39.78	43.25	37.70	47.02	26.79	39.78	44.20	38.67	47.14	25.62	40.91	46.40	38.90	47.11
2x6	24.79	37.41	39.55	35.08	44.19	24.19	37.32	39.49	34.49	43.54	23.42	40.21	41.56	36.03	42.25
S+P	24.49	37.22	40.02	35.68	43.67	23.98	39.17	40.53	34.42	43.17	23.22	40.65	41.51	35.81	42.33
13x7	26.63	39.28	42.13	37.70	46.44	26.27	39.22	43.22	38.14	46.21	25.14	40.99	46.37	38.25	44.95
3D JPEG	23.60	38.63	39.51	35.88	44.72										

Table 36 - Lossy compression results (0.0625 bpp) for a 4-level (PET, CT2) and a 5-level (US, CT1, MRI) wavelet decomposition

	Lossy Coding Results for 0.0625 bpp (in dB)														
	3D Cube -Splitting					3D SPIHT					3D JPEG2000				
	US	PET	CT 1	CT 2	MRI	US	PET	CT 1	CT 2	MRI	US	PET	CT 1	CT 2	MRI
5x3	23.91	38.84	36.86	34.34	43.06	25.85	39.09	38.26	34.49	41.59	22.06	38.86	41.14	34.50	43.76
S	21.12	32.90	29.80	30.71	39.52	20.60	35.03	30.29	29.32	36.95	20.47	35.62	28.58	28.24	35.36
9x7	25.29	38.49	36.24	35.27	42.23	25.02	38.29	38.22	32.95	39.98	24.70	39.06	41.59	33.90	44.05
9x3	23.77	38.99	37.05	34.45	43.25	25.87	39.12	38.21	34.14	41.91	21.95	39.18	41.28	34.54	43.95
13x11	25.07	38.20	35.94	34.98	41.69	24.60	37.90	38.67	32.40	42.89	24.27	39.04	39.56	33.72	43.92
5x11	23.59	38.80	36.74	35.81	43.00	25.66	38.95	38.87	33.83	41.38	21.53	39.41	41.67	34.43	43.99
2x6	24.26	37.18	33.77	33.67	39.76	23.65	36.97	34.18	32.08	37.31	22.95	38.32	36.83	31.85	38.58
S+P	24.20	37.06	38.00	33.60	42.69	23.49	36.83	37.29	28.79	42.23	22.73	38.45	36.66	32.19	38.64
13x7	25.50	38.53	36.52	35.35	42.32	25.36	38.56	37.70	32.89	40.45	24.67	39.44	41.30	33.72	40.74
3D JPEG	15.53	36.42	33.54	30.85	41.73										

Table 37 - Best performing transform per coding technique and per test image for lossy coding (2 bpp) for a 4-level (PET, CT2) and a 5-level (US, CT1, MRI) wavelet decomposition. The figures depict the deviation in percent from the optimal transform.

	Lossy Coding Results for 2bpp (in %)															
	3D Cube -Splitting					3D SPIHT					3D JPEG2000					
	US	PET	CT 1	CT 2	MRI	US	PET	CT 1	CT 2	MRI	US	PET	CT 1	CT 2	MRI	AV
5x3	-0.47	-2.49	**0.00**	-0.14	-0.46	-0.52	-2.36	**0.00**	**0.00**	-0.65	-0.76	-2.48	-1.51	-0.27	-1.27	-0.89
S	-10.19	-16.11	-10.24	-10.34	-10.44	-11.27	-18.49	-8.14	-9.60	-10.05	-12.80	-13.75	-8.40	-6.23	-11.89	-11.20
9x7	-2.27	-1.21	-2.12	-1.05	-1.21	-1.46	-1.54	-1.14	-1.58	-0.82	-2.89	-1.01	-0.33	-0.24	-0.79	-1.31
9x3	-0.13	-2.43	-0.02	**0.00**	**0.00**	-0.30	-2.33	-0.74	-0.27	-0.57	**0.00**	-2.50	-1.43	-0.19	-0.93	-0.79
13x11	-3.40	-1.89	-2.31	-1.84	-1.74	-2.84	-1.80	-1.81	-2.38	-1.47	-4.12	-0.65	**0.00**	-0.23	-0.96	-1.83
5x11	**0.00**	**0.00**	-1.48	-0.13	-0.34	**0.00**	**0.00**	-0.39	-0.51	**0.00**	-1.15	**0.00**	-0.21	**0.00**	**0.00**	-0.28
2x6	-11.08	-6.97	-5.06	-5.63	-4.24	-10.08	-8.74	-6.21	-5.99	-5.59	-8.09	-7.47	-3.48	-2.73	-7.50	-6.59
S+P	-11.23	-6.64	-5.74	-4.51	-4.48	-10.22	-8.63	-5.21	-6.01	-4.86	-8.56	-3.56	-1.84	-3.40	-3.44	-5.89
13x7	-1.80	-1.03	-2.72	-0.67	-0.73	-1.72	-1.72	-1.15	-1.30	-0.91	-2.59	-0.93	-0.21	-0.17	-0.55	-1.22

Table 38 - Best performing transform per coding technique and per test image for lossy coding (1 bpp) for a 4-level (PET, CT2) and a 5-level (US, CT1, MRI) wavelet decomposition. The figures depict the deviation in percent from the optimal transform.

	Lossy Coding Results for 1bpp (in %)															
	3D Cube -Splitting					3D SPIHT					3D JPEG2000					
	US	PET	CT 1	CT 2	MRI	US	PET	CT 1	CT 2	MRI	US	PET	CT 1	CT 2	MRI	AV
5x3	-3.19	-1.43	-0.40	-0.21	-0.70	**0.00**	-0.79	-0.98	-0.15	-0.56	-0.18	-1.77	-2.27	**0.00**	-0.80	-0.89
S	-16.91	-14.18	-12.58	-11.59	-12.21	-18.69	-15.72	-11.01	-10.58	-11.83	-14.76	-10.17	-10.62	-7.55	-14.30	-12.85
9x7	-1.02	-1.24	-1.87	-1.42	-1.51	-2.20	-1.81	-1.05	-1.63	-1.27	-2.56	**0.00**	-0.40	-0.45	-0.53	-1.26
9x3	-2.78	-1.24	**0.00**	**0.00**	-0.44	-0.16	-0.84	-0.83	**0.00**	-0.20	**0.00**	-1.26	-1.92	-0.08	-1.18	-0.73
13x11	-1.94	-1.41	-2.40	-2.19	-2.00	-3.60	-2.43	-1.47	-2.46	-1.26	-3.93	-0.10	**0.00**	-0.72	-0.47	-1.76
5x11	**0.00**	**0.00**	-0.10	-0.42	**0.00**	-0.49	**0.00**	**0.00**	-0.58	**0.00**	-1.07	-0.21	-0.37	-0.05	**0.00**	-0.22
2x6	-15.11	-5.58	-4.77	-5.01	-4.76	-11.91	-7.73	-6.09	-5.77	-5.40	-10.65	-7.51	-5.51	-4.14	-8.86	-7.25
S+P	-16.17	-5.50	-5.31	-4.79	-4.95	-11.50	-7.73	-5.49	-6.16	-5.54	-11.34	-6.36	-2.26	-4.07	-8.16	-7.02
13x7	-0.52	-0.87	-0.89	-1.08	-0.71	-2.25	-1.50	-0.64	-1.29	-0.89	-3.19	-0.03	-0.14	-0.62	-1.24	-1.06

Table 39 - Best performing transform per coding technique and per test image for lossy coding (0.5 bpp) for a 4-level (PET, CT2) and a 5-level (US, CT1, MRI) wavelet decomposition. The figures depict the deviation in percent from the optimal transform.

	Lossy Coding Results for 0.5 bpp (in %)															
	3D Cube -Splitting					3D SPIHT					3D JPEG2000					
	US	PET	CT 1	CT 2	MRI	US	PET	CT 1	CT 2	MRI	US	PET	CT 1	CT 2	MRI	AV
5x3	-0.94	-0.48	-0.64	-0.24	-0.55	-0.87	-0.28	-1.00	-0.11	-1.59	-0.07	-2.04	-1.80	**0.00**	-1.95	-0.84
S	-21.86	-8.37	-13.32	-12.64	-13.92	-16.10	-9.53	-13.07	-7.85	-11.40	-13.91	-10.89	-11.84	-9.94	-18.76	-12.89
9x7	-3.81	-1.49	-1.63	-1.94	-1.47	-4.89	-1.94	-1.25	-2.08	-1.61	-2.32	-0.74	-0.23	-1.09	-0.43	-1.80
9x3	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	-0.98	**0.00**	-0.80	**0.00**	-1.76	-1.38	-0.30	-1.55	-0.45
13x11	-5.04	-2.38	-1.20	-2.85	-2.46	-6.35	-2.86	-2.05	-3.12	-2.49	-3.22	-0.29	**0.00**	-1.33	**0.00**	-2.38
5x11	-1.06	-0.16	-0.26	-0.70	-0.20	-1.24	-0.21	**0.00**	-0.76	**0.00**	-1.07	-0.59	-0.59	-0.66	-1.29	-0.59
2x6	-9.83	-5.36	-5.66	-5.11	-5.62	-11.53	-7.35	-7.19	-6.30	-7.96	-10.48	-5.28	-5.45	-5.38	-10.29	-7.25
S+P	-10.88	-6.08	-6.20	-6.06	-6.59	-12.57	-7.38	-5.71	-7.23	-7.51	-10.75	-3.10	-2.20	-5.99	-5.56	-6.92
13x7	-2.63	-0.81	-1.35	-1.67	-1.38	-3.75	-1.37	-1.22	-1.84	-1.52	-3.47	**0.00**	-0.18	-0.98	-1.72	-1.59

Table 40 - Best performing transform per coding technique and per test image for lossy coding (0.25 bpp) for a 4-level (PET, CT2) and a 5-level (US, CT1, MRI) wavelet decomposition. The figures depict the deviation in percent from the optimal transform.

	Lossy Coding Results for 0.25 bpp (in %)															
	3D Cube -Splitting					3D SPIHT					3D JPEG2000					
	US	PET	CT 1	CT 2	MRI	US	PET	CT 1	CT 2	MRI	US	PET	CT 1	CT 2	MRI	AV
5x3	-1.12	-0.67	-0.44	-0.33	-3.62	-0.97	-0.32	-2.19	-0.43	-3.75	**0.00**	-1.79	-1.49	**0.00**	-2.38	-1.30
S	-12.92	-12.89	-16.66	-8.81	-11.55	-14.40	-13.06	-18.59	-8.78	-14.19	-11.57	-9.39	-21.34	-12.51	-21.17	-13.86
9x7	-6.07	-0.62	-1.15	-2.59	**0.00**	**0.00**	-0.80	-1.61	-3.15	-1.16	-2.59	-1.16	**0.00**	-0.99	-2.50	-1.63
9x3	**0.00**	-4.14	-0.19	**0.00**	-3.28	-0.26	-0.19	-2.01	**0.00**	-2.84	-0.18	-1.56	-2.58	-0.03	-1.80	-1.27
13x11	-7.65	-0.86	-0.59	-3.74	-0.12	-1.56	-1.49	-1.25	-4.30	-1.44	-3.82	**0.00**	-0.40	-1.58	**0.00**	-1.92
5x11	-2.19	**0.00**	**0.00**	-0.80	-2.87	-1.80	**0.00**	**0.00**	-1.39	**0.00**	-1.63	-0.96	-0.06	-0.31	-1.38	-0.89
2x6	-11.78	-3.90	-6.64	-6.32	-10.15	-12.78	-6.50	-9.61	-8.54	-7.01	-4.56	-3.08	-7.24	-6.81	-14.09	-7.93
S+P	-13.20	-3.29	-5.76	-7.38	-2.99	-7.66	-5.62	-7.66	-9.33	-5.39	-6.04	-2.08	-6.77	-6.98	-7.80	-6.53
13x7	-4.62	-0.08	-1.87	-2.28	-1.28	-2.21	-0.54	-2.06	-2.65	-0.95	-3.48	-0.48	-0.36	-1.09	-2.26	-1.75

Table 41 - Best performing transform per coding technique and per test image for lossy coding (0.125 bpp) for a 4-level (PET, CT2) and a 5-level (US, CT1, MRI) wavelet decomposition. The figures depict the deviation in percent from the optimal transform.

	Lossy Coding Results for 0.125 bpp (in %)															
	3D Cube -Splitting					3D SPIHT					3D JPEG2000					
	US	PET	CT 1	CT 2	MRI	US	PET	CT 1	CT 2	MRI	US	PET	CT 1	CT 2	MRI	AV
5x3	-0.91	-0.68	-0.21	-0.61	-0.37	-1.17	-0.57	-2.43	-0.21	-0.34	**0.00**	-2.32	-2.07	-0.16	-0.96	-0.87
S	-12.15	-8.39	-21.17	-8.80	-14.90	-12.62	-7.89	-20.83	-9.74	-20.96	-8.79	-12.57	-29.19	-14.78	-17.47	-14.68
9x7	-5.02	-2.32	-1.55	-1.54	-1.38	-5.46	-2.86	-2.05	-2.16	-2.46	-2.76	-0.21	-0.51	-1.76	**0.00**	-2.14
9x3	**0.00**	**0.00**	**0.00**	**0.00**	-0.28	**0.00**	**0.00**	-2.11	-0.01	**0.00**	-0.30	-1.30	-2.08	**0.00**	-3.04	-0.61
13x11	-6.44	-3.18	-1.63	-2.74	-2.30	-7.09	-3.96	-1.35	-3.43	-3.33	-4.66	**0.00**	-0.39	-2.43	-0.19	-2.88
5x11	-1.63	-0.80	-0.04	-1.30	**0.00**	-2.03	-0.82	**0.00**	**0.00**	-0.05	-1.07	-1.04	**0.00**	-0.48	-0.37	-0.64
2x6	-10.28	-6.73	-8.58	-8.15	-6.02	-11.54	-6.95	-10.66	-10.82	-7.67	-9.56	-2.74	-10.44	-7.83	-10.64	-8.57
S+P	-11.36	-7.19	-7.49	-6.58	-7.14	-12.28	-2.34	-8.31	-11.01	-8.46	-10.32	-1.68	-10.53	-8.38	-10.47	-8.24
13x7	-3.62	-2.06	-2.62	-1.29	-1.24	-3.91	-2.22	-2.21	-1.37	-2.02	-2.93	-0.86	-0.07	-2.15	-4.93	-2.23

Table 42 - Best performing transform per coding technique and per test image for lossy coding (0.0625 bpp) for a 4-level (PET, CT2) and a 5-level (US, CT1, MRI) wavelet decomposition. The figures depict the deviation in percent from the optimal transform.

	Lossy Coding Results for 0.0625 bpp (in %)															
	3D Cube -Splitting					3D SPIHT					3D JPEG2000					
	US	PET	CT 1	CT 2	MRI	US	PET	CT 1	CT 2	MRI	US	PET	CT 1	CT 2	MRI	AV
5x3	-6.23	-0.40	-2.99	-4.10	-0.44	-0.07	-0.07	-1.58	**0.00**	-3.02	-10.66	-1.49	-1.28	-0.12	-0.66	-2.21
S	-17.17	-15.61	-21.57	-14.25	-8.63	-20.38	-10.45	-22.09	-15.00	-13.83	-17.09	-9.69	-31.41	-18.25	-19.72	-17.01
9x7	-0.85	-1.30	-4.61	-1.52	-2.35	-3.28	-2.13	-1.67	-4.47	-6.78	**0.00**	-0.96	-0.21	-1.84	**0.00**	-2.13
9x3	-6.78	**0.00**	-2.48	-3.80	**0.00**	**0.00**	**0.00**	-1.71	-1.02	-2.28	-11.13	-0.68	-0.95	**0.00**	-0.22	-2.07
13x11	-1.71	-2.02	-5.41	-2.32	-3.62	-4.91	-3.11	-0.53	-6.07	**0.00**	-1.72	-1.01	-5.08	-2.38	-0.30	-2.68
5x11	-7.49	-0.49	-3.30	**0.00**	-0.59	-0.81	-0.43	**0.00**	-1.93	-3.52	-12.82	-0.09	**0.00**	-0.32	-0.13	-2.13
2x6	-4.89	-4.66	-11.13	-5.99	-8.07	-8.58	-5.49	-12.08	-6.99	-13.00	-7.05	-2.86	-11.61	-7.79	-12.42	-8.17
S+P	-5.09	-4.96	**0.00**	-6.17	-1.29	-9.19	-5.85	-4.07	-16.54	-1.54	-7.97	-2.53	-12.03	-6.81	-12.28	-6.42
13x7	**0.00**	-1.18	-3.88	-1.28	-2.16	-1.95	-1.44	-3.02	-4.66	-5.68	-0.10	**0.00**	-0.89	-2.37	-7.51	-2.41

Table 43 - Best performing coding technique per transform and per test image for lossy coding (25 bpp) for a 4-level (PET, CT2) and a 5-level (US, CT1, MRI) wavelet decomposition. The figures depict the deviation in percent from the optimal coding technique.

	Lossy Coding Results for 2bpp (in %)														
	3D Cube -Splitting					3D SPIHT					3D JPEG2000				
	US	PET	CT 1	CT 2	MRI	US	PET	CT 1	CT 2	MRI	US	PET	CT 1	CT 2	MRI
5x3	-2.63	-1.81	-2.46	-1.71	-1.80	0.00	0.00	-1.16	0.00	0.00	-1.12	-2.46	0.00	-1.78	-0.92
S	-1.49	-2.08	-5.86	-4.44	-2.41	0.00	-3.23	-2.38	-2.11	0.00	-2.59	0.00	0.00	0.00	-2.35
9x7	-3.48	-1.35	-5.66	-1.04	-2.37	0.00	0.00	-3.45	0.00	0.00	-2.33	-1.81	0.00	-0.17	-0.27
9x3	-2.51	-1.78	-2.56	-1.31	-1.43	0.00	0.00	-1.97	0.00	0.00	-0.58	-2.51	0.00	-1.44	-0.67
13x11	-3.23	-1.78	-6.16	-1.67	-2.46	0.00	0.00	-4.42	-0.65	-0.21	-2.19	-1.19	0.00	0.00	0.00
5x11	-2.68	-1.68	-5.15	-1.19	-2.33	0.00	0.00	-2.83	0.00	0.00	-2.02	-2.34	0.00	-1.01	-0.31
2x6	-5.01	0.00	-5.51	-3.03	-0.58	-1.29	-0.23	-5.42	-1.87	0.00	0.00	-1.20	0.00	0.00	-2.32
S+P	-4.68	-2.54	-7.74	-1.21	-2.75	-0.94	-2.99	-6.00	-1.21	-1.17	0.00	0.00	0.00	0.00	0.00
13x7	-2.76	-0.99	-6.34	-0.95	-1.87	0.00	0.00	-3.57	0.00	-0.06	-1.76	-1.56	0.00	-0.39	0.00
AV	-3.16	-1.56	-5.27	-1.84	-2.00	-0.25	-0.72	-3.47	-0.65	**-0.16**	**-1.40**	**-1.45**	**0.00**	**-0.53**	-0.76

Table 44 - Best performing coding technique per transform and per test image for lossy coding (1 bpp) for a 4-level (PET, CT2) and a 5-level (US, CT1, MRI) wavelet decomposition. The figures depict the deviation in percent from the optimal coding technique.

	Lossy Coding Results for 1bpp (in %)														
	3D Cube -Splitting					3D SPIHT					3D JPEG2000				
	US	PET	CT 1	CT 2	MRI	US	PET	CT 1	CT 2	MRI	US	PET	CT 1	CT 2	MRI
5x3	-5.51	-1.31	-1.59	-1.21	-1.48	0.00	0.00	-0.70	0.00	-0.24	-2.69	-1.50	0.00	-2.47	0.00
S	-2.40	-4.60	-5.55	-2.92	-1.53	-2.15	-5.68	-2.42	-0.67	0.00	0.00	0.00	0.00	0.00	-2.34
9x7	-1.22	-1.38	-4.86	-0.93	-2.55	0.00	-1.29	-2.64	0.00	-1.23	-2.88	0.00	0.00	-1.44	0.00
9x3	-4.97	-1.07	-1.55	-1.14	-1.34	0.00	0.00	-0.91	0.00	0.00	-2.36	-0.94	0.00	-2.69	-0.50
13x11	-0.72	-1.45	-5.75	-0.87	-3.10	0.00	-1.81	-3.43	0.00	-1.28	-2.86	0.00	0.00	-0.88	0.00
5x11	-1.92	-0.67	-3.17	-0.98	-1.58	0.00	0.00	-1.63	0.00	-0.48	-3.09	-0.74	0.00	-2.09	0.00
2x6	-5.94	0.00	-2.68	-0.35	-0.43	0.00	-1.63	-2.60	0.00	0.00	-1.13	-1.91	0.00	-0.92	-3.20
S+P	-7.56	0.00	-6.45	0.00	-0.48	0.00	-1.71	-5.23	-0.30	0.00	-2.35	-0.77	0.00	-0.75	-2.30
13x7	-0.68	-0.98	-4.16	-0.93	-1.05	0.00	-0.95	-2.49	0.00	-0.13	-3.46	0.00	0.00	-1.95	0.00
AV	-3.44	-1.27	-3.97	-1.04	-1.50	**-0.24**	-1.45	-2.45	**-0.11**	**-0.37**	-2.31	**-0.65**	**0.00**	-1.47	-0.93

Table 45 - Best performing coding technique per transform and per test image for lossy coding (0.5 bpp) for a 4-level (PET, CT2) and a 5-level (US, CT1, MRI) wavelet decomposition. The figures depict the deviation in percent from the optimal coding technique.

	Lossy Coding Results for 0.5 bpp (in %)														
	3D Cube -Splitting					3D SPIHT					3D JPEG2000				
	US	PET	CT 1	CT 2	MRI	US	PET	CT 1	CT 2	MRI	US	PET	CT 1	CT 2	MRI
5x3	-0.63	-1.29	-2.12	-1.21	-1.26	0.00	-0.71	-0.57	0.00	-0.23	-2.88	0.00	0.00	-0.90	0.00
S	-7.38	-0.09	-4.89	-6.23	-4.86	0.00	-0.97	-2.75	0.00	0.00	-1.14	0.00	0.00	-3.25	-7.76
9x7	0.00	-3.58	-4.61	-0.94	-3.67	-0.57	-3.63	-2.38	0.00	-1.77	-1.62	0.00	0.00	-0.01	0.00
9x3	-0.56	-1.11	-1.91	-1.09	-1.28	0.00	-0.72	-0.97	0.00	0.00	-3.66	0.00	0.00	-1.31	-0.16
13x11	0.00	-4.88	-4.42	-1.62	-5.04	-0.82	-4.98	-3.39	-0.82	-3.07	-1.26	0.00	0.00	0.00	0.00
5x11	-0.38	-2.42	-2.94	-1.02	-2.27	0.00	-2.08	-0.79	0.00	0.00	-3.49	0.00	0.00	-0.90	-0.70
2x6	0.00	-2.92	-3.47	0.00	0.00	-1.33	-4.59	-3.18	-0.17	-0.42	-3.81	0.00	0.00	-0.20	-2.36
S+P	0.00	-5.83	-7.21	-0.15	-3.71	-1.35	-6.77	-4.91	-0.32	-2.65	-2.98	0.00	0.00	0.00	0.00
13x7	0.00	-3.63	-4.40	-0.92	-2.31	-0.60	-3.80	-2.41	0.00	-0.39	-3.96	0.00	0.00	-0.14	0.00
AV	-0.99	-2.86	-4.00	-1.46	-2.71	**-0.52**	-3.14	-2.37	**-0.15**	**-0.95**	-2.75	**0.00**	**0.00**	-0.75	-1.22

Table 46 - Best performing coding technique per transform and per test image for lossy coding (0.25 bpp) for a 4-level (PET, CT2) and a 5-level (US, CT1, MRI) wavelet decomposition. The figures depict the deviation in percent from the optimal coding technique.

	Lossy Coding Results for 0.25 bpp (in %)														
	3D Cube -Splitting					3D SPIHT					3D JPEG2000				
	US	PET	CT 1	CT 2	MRI	US	PET	CT 1	CT 2	MRI	US	PET	CT 1	CT 2	MRI
5x3	-0.73	-0.98	-3.32	-1.38	-2.29	0.00	0.00	-2.72	0.00	-0.72	-2.30	-0.65	0.00	-0.05	0.00
S	0.00	-5.25	-0.05	-1.51	0.00	-1.14	-4.85	0.00	0.00	-1.30	-1.19	0.00	-1.39	-4.54	-9.96
9x7	-6.61	-0.91	-5.44	-2.61	-0.56	0.00	-0.47	-3.60	-1.71	0.00	-5.76	0.00	0.00	0.00	-2.04
9x3	-0.32	-4.55	-1.99	-1.48	-2.53	0.00	0.00	-1.45	0.00	-0.37	-3.18	-0.55	0.00	-0.51	0.00
13x11	-6.72	-2.30	-4.52	-3.18	-1.15	0.00	-2.31	-2.85	-2.29	-0.76	-5.47	0.00	0.00	0.00	0.00
5x11	-0.97	-0.62	-4.28	-1.49	-4.53	0.00	0.00	-1.96	-0.61	0.00	-3.07	-0.13	0.00	0.00	-2.06
2x6	-5.01	-2.29	-3.71	-0.49	-5.03	-5.54	-4.33	-4.53	-1.39	0.00	0.00	0.00	0.00	0.00	-8.25
S+P	-6.53	-2.66	-3.30	-1.43	0.00	0.00	-4.41	-2.96	-2.06	-0.78	-1.54	0.00	0.00	0.00	-3.98
13x7	-3.02	-1.05	-5.78	-2.19	-2.04	0.00	-0.89	-3.69	-1.11	0.00	-4.50	0.00	0.00	0.00	-2.00
AV	-3.32	-2.29	-3.60	-1.75	-2.01	**-0.74**	-1.92	-2.64	-1.02	**-0.44**	-3.00	**-0.15**	**-0.15**	**-0.57**	-3.14

Table 47 - Best performing coding technique per transform and per test image for lossy coding (0.125 bpp) for a 4-level (PET, CT2) and a 5-level (US, CT1, MRI) wavelet decomposition. The figures depict the deviation in percent from the optimal coding technique.

	Lossy Coding Results for 0.125 bpp (in %)														
	3D Cube -Splitting					3D SPIHT					3D JPEG2000				
	US	PET	CT 1	CT 2	MRI	US	PET	CT 1	CT 2	MRI	US	PET	CT 1	CT 2	MRI
5x3	0.00	-1.37	-4.99	-2.73	-0.32	-1.31	-1.24	-5.09	-1.11	0.00	-5.42	0.00	0.00	0.00	-0.37
S	0.00	-0.57	-2.54	-0.22	0.00	-1.59	0.00	0.00	0.00	-6.86	-2.70	-2.17	-6.10	-4.58	-2.49
9x7	0.00	-5.05	-7.74	-2.07	-1.92	-1.52	-5.56	-6.22	-1.46	-2.72	-4.06	0.00	0.00	0.00	0.00
9x3	0.00	-1.72	-4.79	-2.29	-0.57	-1.05	-1.71	-4.77	-1.07	0.00	-6.57	0.00	0.00	0.00	-2.78
13x11	0.00	-6.08	-7.93	-2.60	-2.64	-1.74	-6.82	-5.66	-2.08	-3.40	-4.50	0.00	0.00	0.00	0.00
5x11	0.00	-2.75	-6.80	-3.10	-0.24	-1.45	-2.76	-4.74	-0.59	0.00	-5.75	0.00	0.00	0.00	-0.06
2x6	0.00	-6.97	-4.84	-2.62	0.00	-2.45	-7.18	-4.98	-4.26	-1.46	-5.54	0.00	0.00	0.00	-4.38
S+P	0.00	-8.43	-3.59	-0.37	0.00	-2.08	-3.63	-2.38	-3.90	-1.13	-5.19	0.00	0.00	0.00	-3.06
13x7	0.00	-4.17	-9.14	-1.43	0.00	-1.35	-4.32	-6.78	-0.28	-0.50	-5.62	0.00	0.00	0.00	-3.21
AV	**0.00**	-4.12	-5.82	-1.94	**-0.63**	-1.61	-3.69	-4.51	-1.64	-1.79	-5.04	**-0.24**	**-0.68**	**-0.51**	-1.82

Table 48 - Best performing coding technique per transform and per test image for lossy coding (0.0625 bpp) for a 4-level (PET, CT2) and a 5-level (US, CT1, MRI) wavelet decomposition. The figures depict the deviation in percent from the optimal coding technique.

	Lossy Coding Results for 0.0625 bpp (in %)														
	3D Cube -Splitting					3D SPIHT					3D JPEG2000				
	US	**PET**	**CT 1**	**CT 2**	**MRI**	**US**	**PET**	**CT 1**	**CT 2**	**MRI**	**US**	**PET**	**CT 1**	**CT 2**	**MRI**
5x3	-7.49	-0.66	-10.40	-0.45	-1.59	0.00	0.00	-6.99	-0.01	-4.95	-14.65	-0.61	0.00	0.00	0.00
S	0.00	-7.63	-1.62	0.00	0.00	-2.50	-1.66	0.00	-4.52	-6.49	-3.07	0.00	-5.64	-8.05	-10.51
9x7	0.00	-1.48	-12.85	0.00	-4.12	-1.05	-1.99	-8.08	-6.56	-9.24	-2.33	0.00	0.00	-3.87	0.00
9x3	-8.10	-0.47	-10.23	-0.25	-1.60	0.00	-0.14	-7.43	-1.15	-4.65	-15.16	0.00	0.00	0.00	0.00
13x11	0.00	-2.15	-9.14	0.00	-5.08	-1.87	-2.92	-2.25	-7.38	-2.35	-3.17	0.00	0.00	-3.62	0.00
5x11	-8.05	-1.54	-11.82	0.00	-2.26	0.00	-1.16	-6.71	-5.54	-5.95	-16.09	0.00	0.00	-3.87	0.00
2x6	0.00	-2.98	-8.32	0.00	0.00	-2.51	-3.51	-7.21	-4.71	-6.16	-5.36	0.00	0.00	-5.41	-2.97
S+P	0.00	-3.61	0.00	0.00	0.00	-2.95	-4.20	-1.85	-14.32	-1.10	-6.10	0.00	-3.53	-4.20	-9.49
13x7	0.00	-2.31	-11.57	0.00	0.00	-0.55	-2.25	-8.71	-6.98	-4.41	-3.25	0.00	0.00	-4.62	-3.73
AV	-2.63	-2.54	-8.44	**-0.08**	**-1.63**	**-1.27**	-1.98	-5.47	-5.68	-5.03	-7.69	**-0.07**	**-1.02**	-3.74	-2.97

Conclusions

It is shown that the coding principles used for the new image compression standard 2D JPEG2000 and quad-tree techniques are also valid for three-dimensional coding. W-based techniques, combined with an embedded bit-stream and an adaptive context-based arithmetic coder have been combined to develop the CS-EBCOT coder.

The experiments have shown that lossy and lossless performances are kernel dependent. At high bit-rates 9x3, 13x11 and 5x11 wavelet kernels are excelling, while at low bit-rates 9x3, 5x11, 9x7 and 5x3 kernels are the best options. As a conclusion, we can state that the 5x11 kernel seems to be the most stable one, delivering an excellent performance over the complete lossy-to-lossless range.

No gain has been achieved by our attempts to optimize the starting probabilities. These probabilities were extracted from some empirical experiments on a set of test images, choosing the coding pass with the highest number of calls. An elaborate study on strategies to condition the probability models could be subject of further work.

When comparing the CS-EBCOT to other three-dimensional coder like Cube-Splitting, 3D SPIHT and 3D JPEG, CS-EBCOT gave good results at any bit-rate, especially at the lowest bit-rates. For lossless coding, which is a very important feature in medical image compression, CS-EBCOT is outperforming the other coders.

It has not been possible to compare the results of CS-EBCOT with the current JPEG2000 Verification Model, though VM results would probably be better as non-normalized transforms have been used in the CS-EBCOT coder. The inclusion of normalized transforms could be also subject for future work.

To sum up, CS-EBCOT is a promising three-dimensional encoding technique that can be considered as belonging to the state-of-the-art of contemporary multidimensional encoding techniques.

The work of the author has been included in the publication referred as [Sch00b].

Bibliography

[Ada00] M.D. Adams, F. Kossentini, "Reversible integer-to-integer wavelet transforms for image compression: performance evaluation and analysis", *IEEE Trans. on Image Processing*, Vol. 9, No.6, pp.1010-1024, June 2000.

[Bil00] Bilgin, Ali; Zweig, george and Marcellin, Michael W. "Three-dimensional image compression with integer wavelet transforms" Applied Optics, Vol. 39, No. 11, 10th April 2000

[Cal96] A. R. Calderbank, I. Daubechies, W. Sweldens, and B. Yeo, Wavelet transforms that map integers to integers, Technical Report, Department of Mathematics, Princeton University, 1996.

[Kim00] Y.S.Kim, W.A. Pearlman, "Stripe-Based SPIHT Lossy Compression of Volumetric Medical Images for Low Memory Usage and Uniform Reconstruction Quality", Proc. of IEEE Int. Conf. on Acoustics, Speech and Signal Processing (ICASSP 2000), Istanbul, Turkey, June 2000

[Kim99] Y. Kim, W.A. Pearlman, "Lossless volumetric medical image compression", Proc. SPIE Conference on Application of Digital Image Processing XXII, Vol. 3808, pp. 305-312, July 1999

[Mal89] S. Mallat, "A Theory for Multiresolution Signal Decomposition: The Wavelet Representation", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol.11, No.7, pp. 674-693, 1989.

[Mor66] G.M. Morton, "A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing", Internal Report IBM Ltd., Ottawa, Canada, 1966.

[Sch00a] P. Schelkens, J.Barbarien, J. Cornelis, "Volumetric data compression based on cube-splitting", *Proc. of 21st Symposium on Information Technology in the Benelux*, Vol.21, pp.93-100, May 2000.

[Sch00b] P. Schelkens, X. Giro, J. Barbarien, A. Munteanu, J. Cornelis, "Compression of Medical Volumetric Data", JPEG2000 Meeting Arles, France, ISO/IEC JTC1/SC29/WG1 JPEG2000/N1712, July 3-7, 2000.

[Sch00c] P. Schelkens, J. Barbarien, J. Cornelis, "Compression of Volumetric Medical Data based on cube-splitting", *Proc. SPIE Conference on Applications of Digital Image Processing XXIII*, Vol.4115, July 2000.

[Sch00d] P. Schelkens, X. Giro, J. Barbarien, J. Cornelis, "3D Compression of Medical Data Based on Cube Splitting and Embedded Block Coding", ProRISC 2000, Veldhoven, The Netherlands, November 29-30, December 1, 2000 (submitted).

[Swe95] W. Sweldens, "The Lifting Scheme: A New Philosophy in Biorthogonal Wavelet Constructions", *SPIE Conference 1995*, Vol. 2569, pp.68-79, 1995.

[Vm_60] ISO/IEC JTC 1/SC 29/WG 1 WG1N1575, January 28th 2000

[Xio99] Z. Xiong, X. Wu, D.Y.Yun, W.A. Pearlman, "Progressive coding of medical volumetric data using three-dimensional integer wavelet packet transform", Proc. SPIE Conference on Visual Communications, Vol. 3653, pp. 327-335,January 1999